Amr Hany Saleh

The European Master's Program in Computational Logic

Masters Thesis

Constraint Reasoning with Local Search for Continuous Optimization

Dissertação para obtenção do Grau de Mestre em Logica Computicional

Orientador : Jorge Cruz, CENTRIA, Universidade Nova de Lisboa

Júri:

Presidente: Pedro Barahona

Arguentes: Jorge Cruz
Sergio Tessaris

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2014

Constraint Reasoning with Local Search for Continuous Optimization

Declaration of Authorship

I, Amr Hany Saleh, declare that this thesis titled, "Constraint Reasoning with Local Search for Continuous Optimization" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Acknowledgements

I would like to thank and express my highest gratitude to all those who gave me the possibility to complete this thesis. Firstly, I would like to thank my supervisor, Professor Jorge Cruz for his guidance, supervision, valuable suggestions and vision he provided during the course of working on this thesis. I am also very thankful to the EMCL commission for giving me the chance to study in a highly acknowledged and competitive masters program like EMCL.

I would like to thank also my family and all my friends for their support and encouragement during the whole period of my studies, without which, I would never have been able to complete this work. Finally, am very thankful to my close friends who helped me along the journey and gave me all their support.

Abstract

Optimization is a very important field for getting the best possible value for the optimization function. Continuous optimization is optimization over real intervals. There are many global and local search techniques. Global search techniques try to get the global optima of the optimization problem. However, local search techniques are used more since they try to find a local minimal solution within an area of the search space.

In Continuous Constraint Satisfaction Problems (CCSP)s, constraints are viewed as relations between variables, and the computations are supported by interval analysis. The continuous constraint programming framework provides branch-and-prune algorithms for covering sets of solutions for the constraints with sets of interval boxes which are the Cartesian product of intervals. These algorithms begin with an initial crude cover of the feasible space (the Cartesian product of the initial variable domains) which is recursively refined by interleaving pruning and branching steps until a stopping criterion is satisfied.

In this work, we try to find a convenient way to use the advantages in CCSP branch-and-prune with local search of global optimization applied locally over each pruned branch of the CCSP. We apply local search techniques of continuous optimization over the pruned boxes outputted by the CCSP techniques.

We mainly use steepest descent technique with different characteristics such as penalty calculation and step length. We implement two main different local search algorithms. We use "Procure", which is a constraint reasoning and global optimization framework, to implement our techniques, then we produce and introduce our results over a set of benchmarks.

Keywords: Optimization, Constraint Satisfaction, Local search ...

Contents

List of Figures

1

Introduction

In economics, engineering, scientific studies, optimization concepts and tools are used to model quantitative decisions. The aim is always to try to find the "absolute best" decision which corresponds to the minimum (or maximum) of the suitable model's objective. At the same time, a given collection of feasibility constraints might be modelled in the problem. The best decision must be satisfying these constraints. The objective in an optimization model expresses overall system performance, such as profit, loss, risk, or error. The constraints originate from physical, technical, economic or some other considerations.

An optimization problem in mathematical settings is a function representing the objective of the problem over a set of variables having a domain, together with a set of constraints over these variables. Optimization is typically divided into two closely related main research fields. Global optimization is the first research field. It is concerned with finding a global optimal solution for an optimization problem in a mathematical setting.

The second field is the local optimization. Local optimization is the term used for localizing the search of the optimal solution within a part of the search space of the optimization problem.

There are many techniques for solving optimization problems depending on the nature of the problem, whether it contains constraints (constrained optimization) or not (unconstrained optimization). Also depending on the aim of the search, whether to find a local optimum or global optimum.

The concept of constraint satisfaction is very close to optimization. It can be viewed as an optimization problem that does not contain an objective function that needs to be optimized. However, the model contains feasibility constraints which are needed to be

satisfied by the variables of the problem.

Constraint satisfaction problems and their techniques are divided mainly into two parts: *CSP*, which is the normal *C*onstraint *S*atisfaction *P*roblem (models) in discrete mathematical settings, and *CCSP*, which is the constraint satisfaction problem in *C*ontinuous settings. Continuous means that the domains of the variables of the problem are infinite. In this thesis, we focus more on *CCSP*.

With these two very brief introductions, we can observe that constraint satisfaction problems and optimization are very close in their definitions. Constraint satisfaction seeks a feasible solution, and optimization seeks an optimal solution. An optimization problem can be defined as a constraint satisfaction problem with an additional optimization function that should be optimized along with satisfying the constraints in the problem. Therefore, there is a potential of using techniques from both fields and combine them.

In the recent years, hybrid models that use optimization with constraint satisfaction appeared, as it was shown in [22]. The recent interaction between optimization and constraint satisfaction promises to change both fields. It might be in the near future that portions of both will merge into a single problem-solving technology for discrete and continuous problems.

In this work we present two techniques that combine local search optimization with continuous constraint satisfaction. The main aim is to find a global optimal solution by dividing the search space using constraint satisfaction's branching techniques into areas (boxes), and apply local search on these areas, keeping track of the optimal solution found so far. For each divided area in the search space, *CCSP* pruning techniques are applied, after which we use one of the two local search techniques, in order to find the local optimal solution of this search area.

The first technique is Armijo Rule with Quadratic Penalty Steepest Descent, and the second is Box Ratio with Separate Penalty Steepest Descent. We make comparisons between these two techniques, trying to find a conclusion of which is having better timing and suitability for the given model. Moreover, we try to investigate the quality of the local search in every box, as well as the convergence and stopping strategies used.

The rest of the work is organized as follows: in chapter 2, we discuss the continuous optimization techniques in mathematical settings. We present a survey on the current techniques of global and local optimization. Focusing more on the specific techniques that we use to implement our hybrid local search with continuous constraint satisfaction *(CCSP-LS)* algorithms.

Afterwards, in chapter 3, we show the necessary theory needed for this work about constraint satisfaction techniques over continuous domains. First, we introduce interval analysis and interval function, since the theory of *CCSP* is built over these concepts. Then we discuss briefly the concept of constraint propagation that is used to prune the search space. We also discuss consistency techniques used in constraint satisfaction.

After presenting the background needed for this work, we introduce the two hybrid

CCSP-LS techniques in chaper 4. We first introduce the generic algorithm for *CCSP-LS*. Then, in section 4.1, we give a brief overview of “Procure", the framework we use in the implementation of the algorithms. In section 4.3 we show the first hybrid technique and in section 4.4 we discuss the second technique.

In chapter 5, we show the tests we conducted on the algorithms. We run the algorithms over two main examples, getting information of the pruning speed and the time taken by local search techniques to find a local optimum. Moreover, we run the algorithms on a set of benchmarks, and show the time and comparisons between the different techniques.

In chapter 6, we discuss the results of the experiments conducted in chapter 5. We compare the different techniques and point out the strength and weakness of every technique. Moreover, we suggest future work ideas to investigate further the research topic.

2

Continuous Optimization

Optimization is important in science, mathematics and everyday problems. It dates back to 300 B.C., when Euclid considered the minimal distance between two points to be a line, and proved that a square has the greatest area among the rectangles, given the total length of edges.

In order to use optimization, we need to identify an *objective* to a given problem, in other words, a *model*. The objective depends on certain characteristics of the system, called *variables*. The goal of the objective is to find values of the variables that optimize the objective. The variables in the given model might be constrained, meaning that they have restrictions over their domains. In this thesis, we deal with optimization in mathematical settings with variables having continuous domains.

In mathematics, an optimization problem is the minimization or maximization of an objective function f over a vector of variables x. This is subject to a vector of constraints c that the variables in x must satisfy.

An optimization problem can be written as:

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{subject to} \quad \begin{cases} c_i(x) = 0 & 1 \leq i \leq k \\ c_i(x) \leq 0 & k < i \leq m \end{cases} \tag{2.1}$$

such that,

- $x = (x_1, x_2, \ldots, x_n)$ is a vector of n variables of the problem.
- A function $f : \mathbb{R}^n \to \mathbb{R}$ which is the objective function to be minimized.
- Constraints $\{c_i(x) = 0 \quad |1 \leq i \leq k\}$ are equality constraints over the variables in the vector x;

- Constraints $\{c_i(x) \leq 0 \quad |k < i \leq m\}$ are inequality constraints over the variables in the vector x;

Transformation of the equations in the given model is often necessary to express an optimization problem into the standard form shown above. A very common example is changing *maximization* problems into *minimization* problems by negating the objective function f to $-f$.

An assignment of values to all variables in x represents an (candidate solution) *option*. The constraints represent limitations on the *options*. A feasible option is a solution that does not violate any constraint c_i. The objective function represents the cost d, $d = f(x)$. The set of all feasible options is called the solution space. A *global optimum*, of the problem, x^*, which is also called a *global minimizer* since the standard optimization problem is a minimization problem, is a feasible solution $x^* = (x_1^*, x_2^*, \ldots, x_n^*)$, whose cost is less than or equal to any solution belonging to the space of solutions.

$$\forall x \quad f(x^*) \leq f(x) \tag{2.2}$$

Normally, finding a global optimum is not as easy as finding a local optimum. A local optimum location of the problem, also noted as x^*, is a feasible solution whose cost is less than or equal to any other feasible solution belonging to a neighborhood $\mathcal{N}$, which is a subset of the search space S of the problem; $\mathcal{N} \subseteq S$. A neighborhood $\mathcal{N}$ of x^* is an open set of vectors that contains x^*.

$$\forall x_{x \in \mathcal{N}} \quad f(x^*) \leq f(x) \tag{2.3}$$

Most of the optimization algorithms in use today are iterative. They have a solid theoretical basis, but the theory often allows wide latitude in the choice of certain parameters, and algorithms are often "engineered" to find suitable values for these parameters and to incorporate other heuristics.

Analysis of algorithms tackles such issues as whether the iterates can be guaranteed to converge to a solution; whether there is an upper bound on the number of iterations needed, as a function of the size or complexity of the problem; and the rate of convergence, particularly after the iterates enter a certain neighborhood of the solution.

Algorithmic analysis is typically worst-case in nature. It gives important indications about how the algorithm will behave in practice, but does not tell the whole story. A key aspect in solving optimization problems is the recognition of optimal solutions. Under certain assumptions, derivatives of the objective function and constraints can be used to define a set of test conditions to verify if a candidate solution is in a good place with respect to the objective function.

However it is not easy to check global optimum, even when the objective functions and constraints are differentiable, due to the difficulty in obtaining an overview of these

functions. In the particular case of convex optimization problems, where both the objective function as well as the solution space is convex, meaning all local optima are necessarily a global optimum. For the general case of non-convex problems, several global optimization algorithms have been proposed, some of which use techniques of integrated global local search procedures exploring the space of solutions in partitions.

Often, the variables in the model are constrained. However, some optimization problems are constraints-free. This leads to dividing the optimization problems into *constrained* and *unconstrained* optimization. Moreover, optimization algorithms are mainly divided into *local search* algorithms, which targets finding the *local optimum* and *global search* algorithms, which targets finding the *global optimum*,

Section 2.1 is dedicated to local search algorithms and their strategies to take advantage of the characteristics of different types of optimization problems. Afterwards, in section 2.2, global optimization algorithms is presented, whose main objective is to achieve global optimal solutions and therefore should introduce strategies to prevent their termination in local optimal places (with values of the objective function significantly different from the global optimum).

2.1 Local Search Optimization

Local search algorithms are iterative. They begin with an initial guess, whether it is a random guess or a point supplied by the user who has knowledge about the application and the data set, and may be in a good position to choose a reasonable estimate of the solution. Beginning at the starting point, the algorithm generates a sequence of iterations to try to find points with improved estimates. The algorithm terminates when either no more progress can be made or when it seems the point in the final iteration has been approximated with sufficient accuracy.

A local search algorithm is distinguished from the others by the strategy of deciding how to move from one iteration to the next. The algorithm uses information about the objective function $f(x)$ and the constraints $c_i(x)$ and their derivatives, possibly combined with information gathered at earlier stage iterations of the algorithm.

In this section, we will give an overview of the local search algorithms for *unconstrained* and for *constrained* optimization problems. For more detailed discussion of each of these approaches, we suggest reading the classical references [50, 12, 17] in this area.

2.1.1 Unconstrained Local Search

Unconstrained optimization problem is defined by optimizing an objective function with no restrictions on the values of the variables involved in the function. The standard mathematical formula for the unconstrained optimization problem is:

$$\min_{x \in \mathbb{R}^n} f(x) \tag{2.4}$$

which is an instance of equation 2.1 with $m = 0$. There are two main strategies for unconstrained optimization; *Line Search* and *Trust Region*.

In *Line Search* strategy [36], the algorithms depend on two main factors to obtain a new vector x_{k+1} from the current iteration vector x_k: a direction d_k and a step α. The general mathematical form of the *Line Search* algorithm is:

$$x_{k+1} = x_k + \alpha_k d_k \quad \text{such that} \quad \alpha_k > 0 \tag{2.5}$$

At each iteration, a search direction d_k and a positive scalar α_k are decided. α_k decides how far to move along d_k. Therefore, *line search* strategy is mainly divided into two main criteria: the selection of the *step length* and deciding over the *direction*.

The selection of the step length always faces a trade-off between finding the best scalar value that would give a substantial reduction of f, and the computation time to get such value. Therefore, line search algorithms try out a sequence of values of α_k and stop when certain conditions are satisfied.

A very popular algorithm for calculating the *step length* is the *exact line search* algorithm which has the following formulation:

$$\alpha_k = \min_{\alpha} f(x_k + \alpha d_k), \tag{2.6}$$

with the following stopping condition:

$$f(x_k + \alpha_k d_k) < f(x_k). \tag{2.7}$$

Another popular algorithm for calculating *step length* is the *inexact line search* algorithm. It has the following stopping condition:

$$f(x_k + \alpha d_k) \leq f(x_k) + \mathbf{c_1} \alpha \nabla f_k^{\mathsf{T}} d_k \tag{2.8}$$

Typically, $0 < \mathbf{c_1} \leq 1$ is a scalar factor. ∇f_x is the vector of partial derivatives of f over the variables in the vector x, which is also called the gradient of f at the point x. The reduction in f is proportional to both α and the directional derivative multiplied by the direction: $\nabla f_k^{\mathsf{T}} d_k$. This method is also called the *Armijo rule*, named after Larry Armijo [2].

Many popular optimization algorithms are based on the *line search* strategy. For example, one of the very intuitive algorithms is *the steepest descent* algorithm, mentioned in [37, 1]. It sets the direction d_x to be the gradient of the objective function, ∇f, and it has the following formulation on obtaining x_{k+1} from x_k:

$$f(x_{k+1}) = f(x_k) - \alpha_k \nabla f_k \tag{2.9}$$

Steepest descent uses the directional derivative ∇f_k to get the direction of the gradient of the objective function at the point x_k. Then, since the problem is a minimization

problem, we take the opposite direction of the gradient $-\nabla f_k$.

Other popular line search algorithms include *The Newton*, *The Quasi-Newton* and *Conjugate Gradient* algorithms.

In the *Trust Region* strategy [35, 7], the collected information about f is used to construct a model function m_k, whose behavior near the current point x_k is similar to that of the actual objective function f. Because the model m_k may not be a good approximation of f when x^* is far from x_k, we restrict the search for a minimizer of m_k to some region around x_k. We try to find a candidate step p by approximately solving the following subproblem:

$$\min_p m_k(x_k + p) \quad \text{where} \quad x_k + p \quad \text{lies inside the trust region} \tag{2.10}$$

If p does not give a sufficient decrease in the objective function f, then the trust region is too big. Therefore, it is shrinked and the subproblem is reevaluated. The trust region has a radius Δ and the step p is normally bounded by it; $||p||_2 \leq \Delta$. The definition of the model for m_k is usually a quadratic function having the following formulation:

$$m_k(x_k + p) = f_k + p^\intercal \nabla f_k + \frac{1}{2} p^\intercal B_k p, \tag{2.11}$$

where f_k and ∇f_k are the objective function and its gradient values at the point x_k, respectively. B_k is usually the second derivative of f_k or an approximation to it.

The trust region algorithm technique is to first choose a maximum trust-region radius Δ_k, and then seek a direction and step that gets the best improvement possible subject to this region constraint. If this step proves to be unsatisfactory, we reduce the radius measure Δ_k and try again.

There are several popular trust region algorithms including the *Cauchy point* algorithm, the *Dogle* method and the *Steihaug's approach* [45].

2.1.2 Constrained Local Search

Constrained optimization is the normal case of optimization problem, having the standard minimization formulation, which we saw in equation 2.1. The set Ω of feasible points that are candidate solutions for the optimization problem 2.1 is defined by:

$$\Omega = \{x \in \mathbb{R}^n | c_i(x) = 0, \quad 0 \leq i \leq k; \quad c_i(x) \leq 0, \quad k < i \leq m\} \tag{2.12}$$

Such that n is the number of variables in vector x. Equation 2.1 can be rewritten to:

$$\min_{x \in \Omega} f(x) \tag{2.13}$$

Constrained Local search can be divided into two main categories depending on the nature of the objective function f and the constraints $c_i(x)$. These parts are *Linear programming* and *Non-linear constraint optimization*.

Linear programming [25], which dates back at least as far as Fourier, is an important special case of constrained optimization problems for which very efficient specialized algorithms are available. These problems are specified by having a linear objective function f and linear constraints c_i.

In these specific problems, the contours of the objective functions are planner. Therefore, a local minimum or global minimum must lie on a vertex of the feasible set.

There are two important algorithms for linear programming: *The Simplex Method* and *The Interior Point Method*.

The simplex method which was developed by George Dantzig [9] moves from vertex to neighboring vertex of the feasible set, decreasing the objective function with each move, and terminating when it cannot find a neighboring vertex with a lower objective value. It mainly depends on maintaining the KarushKuhnTucker (KKT) condition and performing some operations on the feasible set if one of the *KTT* conditions got violated.

Interior-point methods, introduced by Karmarkar [26], approach the boundary of the feasible set only in the limit. They may approach the solution either from the interior or the exterior of the feasible region, but they never lie on the boundary of this region. Each interior-point iteration is expensive to compute and can make significant progress toward the solution, while the simplex method usually requires a larger number of inexpensive iterations.

Geometrically speaking, the simplex method works its way around the boundary of the feasible polytope, testing a sequence of vertices in turn until it finds the optimal one. Interior-point methods approach the boundary of the feasible set only in the limit. They may approach the solution either from the interior or the exterior of the feasible region, but they never actually lie on the boundary of this region.

Non-Linear Constraint Optimization category is what mainly most constrained optimization problems classification fall under. It is an optimization problem where the objective function f or one of the constraints $c_i(x)$ are not linear.

There are many algorithms and approaches developed for solving non-linear constraint optimization problems. However, most of them depend on transforming or simplifying the problem into subproblems, that have efficient algorithms to solve them.

Very popular techniques for solving non-linear constraint optimization problems are the penalty and augmented Lagrangian methods. These techniques' idea is mainly to transform the constrained problem into unconstrained problems by applying penalty value for the constraints involved in the problem, generating a new objective function that includes these penalty values. One of the very popular algorithms is the *quadratic penalty method*, mentioned in [50], which adds to the objective function an additional term for every constraint in the problem. The newly obtained objective function $Q(x)$ has the following formula:

$$Q(x) = f(x) + \frac{1}{2\mu}\sum_{i=0}^{i=k} c_i^2(x) + \frac{1}{2\mu}\sum_{i=k+1}^{i=m} \max(c_i(x), 0)^2 \tag{2.14}$$

Such that $\mu > 0$ is the penalty parameter controlling the impact of the penalty values over the obtained objective function Q. Afterwards, we try to minimize the unconstrained objective function Q using the algorithms for unconstrained optimization problems with a series of increasing values of μ.

Another penalty oriented method is the augmented Lagrangian method, where we define a function that combines the properties of the quadratic penalty function of the Lagrangian function. The Lagrangian function for the standard optimization problem defined in (2.1) is defined as follows:

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i=0}^{i=m} \lambda_i c_i(x), \tag{2.15}$$

where λ_i is a vector of Lagrangian multipliers. This so-called augmented Lagrangian function has the following form for equality-constrained problems:

$$\mathcal{L}_A(x, \lambda; \mu) = f(x) - \sum_{i=0}^{i=k} \lambda_i c_i(x) + \frac{\mu}{2} \sum_{i=0}^{i=k} c_i^2(x) \tag{2.16}$$

We try to fix the value of λ to some estimate of the optimal Lagrange multiplier vector, then fix μ to a value greater than zero, then find a value of x that approximately minimizes $\mathcal{L}_A(x, \lambda; \mu)$. The augmented Lagrangian method was first introduced by Hestences [21] and Powell [39]. It was also mentioned in [51].

From the other several algorithms available for non-linear constraint optimization problems, there are also the *sequential quadratic programming*; introduced in late 1970's, described in [5]. It can be used both in line search and trust-region frameworks, and it is appropriate for both small or large problems. Here the idea is to model the original problem by a quadratic subproblem at each iteration, and to define the search direction as the solution of this subproblem. The objective in this subproblem is an approximation of the Lagrangian function and the constraints are linearizations of the original constraints. The new iteration is obtained by searching along this direction until a certain merit function is decreased. Sequential quadratic programming methods have proved to be effective in practice. They are the basis of some of the best software for solving both small and large constrained optimization problems. They typically require fewer function evaluations than some of the other methods, at the expense of solving a relatively complicated quadratic subproblem at each iteration.

2.2 Global Search Optimization

Obtaining global optimum in problems with constraints in continuous domains requires the use of strategies that are able to seek solutions in the search space without getting trapped in local minima. One of the famous reference books of global optimization is written by Horst et al. [23].

One class of methods for solving the global optimization problem are the deterministic methods. One of the techniques of the deterministic methods, in other words: exact methods, is to use a process of subdividing the feasible region and using information about the objective function to obtain a lower bound on the objective in that region, leading to the branch-and-bound algorithm. These methods can provide assurance of the quality of the solution found.

There is also another class of methods which depends on meta-heuristics. However, this class of methods does not guarantee the quality of the solution.

In this section, we will give a brief introduction to the the different classes of global optimization algorithms.

2.2.1 Deterministic methods

Deterministic algorithms for continuous global optimization problems guarantee that by the termination of it to have found at least one global minimum of the given optimization problem. However, since the domains of the variables are continuous, the algorithms are more likely to take a very long time to find a global minimum.

There are several deterministic global optimization algorithms. We will discuss the *branch-and-bound* algorithms, discussed in [19, 27]. These algorithms are considered the most popular deterministic global optimization methods. To apply branch-and-bound, one must have a means of computing a lower bound on an instance of the optimization problem and a means of dividing the feasible region of a problem to create smaller subproblems. There must also be a way to compute an upper bound (feasible solution) for at least some instances; for practical purposes, it should be possible to compute upper bounds for some set of nontrivial feasible regions. The method starts by considering the original problem with the complete feasible region, which is called the *root problem*.

The lower-bounding and upper-bounding procedures are applied to the root problem. If the bounds match, then an optimal solution has been found and the procedure terminates. Otherwise, the feasible region is divided into two or more regions, each strict subregions of the original, which together cover the whole feasible region, these subproblems partition the feasible region. These subproblems become children of the root search node.

The algorithm is applied recursively to the subproblems, generating a tree of subproblems. If an optimal solution is found to a subproblem, it is a feasible solution to the full problem, but not necessarily globally optimal. Since it is feasible, it can be used to prune the rest of the tree: if the lower bound for a node exceeds the best known feasible solution, no globally optimal solution can exist in the subspace of the feasible region represented by the node. Therefore, the node can be removed from consideration. The search proceeds until all nodes have been solved or pruned, or until some specified threshold is met between the best solution found and the lower bounds on all unsolved subproblems.

Some other strategies include *Bayesian search algorithms*, introduced by Jonas Mockus

[32]. In Bayesian search algorithms, techniques for modeling data by Bayesian networks are developed to estimate the joint distribution of promising solutions. Moreover, *enumerative methods* which, as its name suggests, enumerates all possible solutions. However, since the domains of the variables are infinite in continuous domains, a set of approximates of all the possible solution is used. This strategy is applicable to small sized problems which have a small feasible area.

2.2.2 Meta-heuristic methods

Meta-heuristic methods do not guarantee to obtain the global optimum solution for the optimization problem. However, the techniques followed by these methods make them reach a solution which is very close to the local optimum or even the global optimum itself. Moreover, the meta-heuristic methods are very competitive to the deterministic methods, because it takes much less time to compute and give very good results. Meta-heuristic methods are discussed in details in [16].

Local search methods can get stuck in a local minimum, where no improving neighbors are available. A simple modification consists of iterating calls to the local search routine, each time starting from a different initial configuration. This is called repeated local search, and implies that the knowledge obtained during the previous local search phases is not used. Learning implies that the previous history, for example the memory about the previously found local minima, is mined to produce better starting points for local search. *Iterated Local Search* algorithm, described in [29], is based on building a sequence of locally optimal solutions by perturbing the current local minimum and applying local search after starting from the modified solution.

Another meta-heuristic method is *Simulated Annealing*; discussed in [47, 11] which is a random-search technique which exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process), and the search for a minimum in a more general system. The major advantage over other methods is an ability to avoid becoming trapped in local minima.

The algorithm employs a random search, which not only accepts changes that decrease the objective function, but also some changes that increase it. The latter are accepted with a probability given by a function of the increase in the objective function and a control parameter, known as the system "temperature". The temperature is decreasing (cooling) slowly according to a predefined schedule, in order to decrease the probability of accepting worst solutions as exploring the space of solutions.

There is also the *Tabu Search* method, discussed in [6]. The idea of this popular meta-heuristic algorithm is to block the search moves to points already visited in the search space, or just block it for the next k iterates. The main use of the tabu search algorithm is in discrete optimization problems, but it can also be extended to handle continuous global optimization problems.

Evolutionary Algorithms (EA) [31] are search methods that take their inspiration from

natural selection and survival of the fittest in the biological world. EA differ from more traditional optimization techniques in that they involve a search from a “population" of solutions, not from a single point. Each iteration of an EA involves a competitive selection that weeds out poor solutions. The solutions with high “fitness" are “recombined" with other solutions by swapping parts of a solution with another. Solutions are also “mutated" by making a small change to a single element of the solution. Recombination and mutation are used to generate new solutions that are biased towards regions of the space for which good solutions have already been seen.

Several different types of evolutionary search methods were developed independently. These include (a) Genetic Programming (GP), which evolve programs, (b) Evolutionary Programming (EP), which focus on optimizing continuous functions without recombination, (c) Evolutionary Strategies (ES), which focus on optimizing continuous functions with recombination, and (d) Genetic Algorithms (GAs), which focus on optimizing general combinatorial problems.

3

Constraint Satisfaction over Continuous Domains

Constraints are the way to specify a relation that must hold between two or more variables. This is done by restricting the possible values that these variables can have. In mathematics, constraints are accurately specified relations between variables. A **C**onstraint **S**atistfaction **P**roblem (CSP) is a model with variables, such that each variable is ranging over a domain. Constraints over these variables restrict the respective domain values that can be assigned to the variables. The aim of any CSP is to give a value for each variable that does not violate any of the constraints. CSP was introduced in early seventies in [49, 33, 30]. The following definition is the formal definition of a constraint, mentioned in [8][1]:

Definition 3.1. A constraint c is a pair (s, ρ), where s is a tuple of m variables $< x_1, x_2, \ldots, x_m >$, the constraint scope, and ρ is a relation of arity m, the constraint relation. The relation ρ is a subset of the set of all m-tuples of elements from the Cartesian product $D_1 \times D_2 \times \cdots \times D_m$ where D_i is the domain of the variable x_i:

$$\rho \subseteq \{< d_1, d_2, \ldots, d_m > | d_1 \in D_1, d_2 \in D_2, \ldots, d_m \in D_m\} \tag{3.1}$$

The tuples in ρ are the tuples that allow the satisfaction of c. The arity of c is m, which is the length of the tuples in ρ.

[1] Most of the definitions mentions in this chapter were also mentioned in this reference

The constraints in a CSP can be represented explicitly, by stating all the allowed combinations of variables' values of every constraint c in the CSP. They can also be represented implicitly by means of mathematical expressions or procedures that can be computed in order to determine these combinations.

Definition 3.2. A CSP is a triple $P = (X, D, C)$ where X is a tuple of n variables $< x_1, x_2, \ldots, x_n >$, D is the Cartesian product of the respective domains $D_1 \times D_2 \times \cdots \times D_n$, i.e. each variable x_i ranges over the domain D_i, and C is a finite set of constraints where the elements of the scope of each constraint are all elements of X.

A solution to a CSP is a tuple of values, each value assigned to a variable, such that it satisfies all the constraints in C.

Definition 3.3. A solution to the CSP $P = (X, D, C)$ is a tuple $d \in D$ that satisfies each constraint $c \in C$, that is:
d is a solution of P iff[2] $\forall_{c_i=(s_i,\rho_i)\in C}\ d \in \rho_i$

A CSP may have one, several, or no solutions. In practice, the modeling of a problem as a CSP is embedded in a larger decision process. Depending on this decision process it may be desirable to determine whether a solution exists, meaning that the overall CSP is consistent. The process might also try to find a solution for the CSP, a set of whole solutions to the CSP or an optimal solution for an objective function (to turn into an optimization problem).

Definition 3.4. A CSP, $P = (X, D, C)$ is consistent iff it has at least one solution: P is consistent iff $\exists d \in D$ such that d is a solution of P.

There are several ways to solve a CSP. The main strategy is to use each constraint separately and try to eliminate values of the variables that guaranteedly can not satisfy it (and consequently, no valid solution is lost). This is called *constraint propagation*. This decreases the search space where the algorithm is looking for a solution. We discuss constraint propagation more in section 3.2.

Numeric CSP's, introduced by Davis in [10], are the extension of the CSP to include variables over continuous domains, since earlier CSPs were only devoted for problems including variables over finite domains.

Definition 3.5. A NCSP is a CSP $P = (X, D, C)$ where:
i) $\forall_{D_i\in D} D_i \subseteq \mathcal{Z} \vee D_i \subseteq R$
ii) $\forall_{(s,\rho)\in C}$ ρ is defined as a numeric relation between the variables of s.

In NCSP, constraints have to be expressed implicitly, as a numeric relation between the variables, since the explicit representation of the constraints might be infinite.

Continuous CSPs are a special class of NSCP where the shape of constraints expression has to be specific. This is the definition of CCSP based on [41].

[2] if and only if

Definition 3.6. CCSP is a CSP, $P = (X, D, C)$ where each domain is an interval of $\mathbb{R}$ and each constraint relation is defined as a numerical equality or inequality:
i) $D =< D_1, \ldots, D_n >$ where D_i is a real interval $(1 \leq i \leq n)$.
ii) $\forall_{c \in C} \quad c$ is defined as $e_c \diamond 0$ where e_c is a real expression and $\diamond \in \{\leq, =, \geq\}$.

In a CCSP, the domains associated with the variables are intervals which are infinite sets of real numbers. However, to represent infinite sets of real numbers on a computer system, several techniques were developed. In section 3.1 we talk about how CCSP domains are represented and operations that can be done over them. Afterwards, in section 3.2, we discuss the concept of constraint propagation and the elimination of inconsistent values. In section 3.3, we discuss popular consistency techniques for CCSPs.

3.1 Interval Representation and Analysis

In order to represent a continuous domain in a computer system, *F*-numbers were presented, as defined in several publications [28, 3, 46].

Definition 3.7. Let **F** be a subset of $\mathbb{R}$ containing the real number 0 as well as finitely many other reals, and two elements (not reals) denoted by $-\infty, +\infty$:
$\mathbf{F} = r_0, \ldots, r_n \cup \{-\infty, +\infty\} \quad$ with $\quad 0 \in \{r_0, \ldots, r_n\} \subset \mathbb{R}$
The elements of **F** are called F-numbers.

The elements of **F** are totally ordered wrt[3] $\mathbb{R}$. Moreover, if f is an *F*-number, then f^- and f^+ are two *F*-numbers which are directly before and after f in **F** wrt the total order.

With the introduction of *F*-number, *F*-interval is introduced, which is the subset of real intervals that can be represented by a particular machine as the set of real intervals bounded by *F*-numbers.

Definition 3.8. An *F-interval* is a real interval $\emptyset$ or $< a..b >$ where a and b are *F*-numbers. In particular, if $b = a$ or $b = a^+$ then $< a..b >$ is a canonical *F*-interval.

To express a set of variables' domains in a CCSP, we use the notion of a box. An *F-box* is an extension to the concept of *F*-interval, with several dimensions.

Definition 3.9. A *F-box BF* with arity n is the Cartesian product of n *F*-intervals and is denoted by $< IF_1, \ldots, IF_n >$ where each IF_i is an *F*-interval:
BF = $\{< r_1, r_2, \ldots, r_m > \quad | \quad r_1 \in IF_1, r_2 \in IF2, \ldots, r_n \in IF_n\}$.
In particular, if all the *F*-intervals IF_i are canonical, then BF is a canonical *F-box*.

Interval analysis, introduced in [34], is very important for CCSP in order to be able to eliminate inconsistent solutions. It is used in many proofs of the soundness of constraint propagation techniques. Interval analysis is based on interval arithmetic, which is an extension of real arithmetic for real intervals.

[3] with respect to

Interval arithmetic redefines the basic real arithmetic, like sum, difference, product and quotient. We define the basic interval arithmetic operators as follows:

Definition 3.10. Let I_1 and I_2 be two real intervals. The basic arithmetic operations on intervals are defined by:

$$I_1 \Phi I_2 = \{r_1 \Phi r_2 \quad |_1 \in I_1 \wedge r_2 \in I_2\} \quad \text{with} \quad \Phi \in \{+, -, \times, /\}$$
$$\text{except that} \quad I_1/I_2 \quad \text{is not defined if } 0 \in I_2. \quad (3.2)$$

To define the basic operators, let $[a..b]$ and $[c..d]$ be two real intervals. Therefore:

- $[a..b] + [c..d] = [a + c..b + d]$
- $[a..b] - [c..d] = [a - d..b - c]$
- $[a..b] \times [c..d] = [\min(ac, cd, bc, bd).. \max(ac, cd, bc, bd)]$
- $[a..b/[c..d] = [a..b] \times [1/d..1/c] \quad$ if $0 \notin [c..d]$

Most algebraic properties in the case of real arithmetic, such as commutativity and associativity hold also in interval arithmetic, However, in distributivity, if we have I_1, I_2 and I_3 intervals then the sub-distributivity law becomes:

$$I_1 \times (I_2 + I_3) \subseteq I_1 \times I_2 + I_1 \times I_3 \quad (3.3)$$

According to [34], interval analysis and evaluations are sound and correct. This soundness is one of the major contributions to the interval constraints framework. Since a function can be expressed in different equivalent expressions, interval evaluations of these expressions may yield different interval results. However, the soundness of interval arithmetic guarantees that all the output intervals contain the intended results for the function. We show the formal definition of interval expressions and the representation of interval function next.

Definition 3.11. An expression E is an inductive structure defined in the following way:
(i) a constant is an expression;
(ii) a variable is an expression;
(iii) if $E_1, \ldots, E_m$ are expressions and ϕ is a m-ary basic operator then $\phi(E_1, \ldots, E_m)$ is an expression; a real expression is an expression with real constants, real-valued variables and real operators. An interval expression is an expression with real interval constants, real interval valued variables and interval operators.

An interval constraint is represented by an interval expression. There are many algorithms and techniques for interval analysis. This is described more in detail in [34].

3.2 Constraint Propagation

In a CCSP $P = (X, C, D)$, the initial domains of variables in X are infinite sets, since they are real intervals. The whole domain of the problem obtained by the power set of D is also infinite. Therefore, solving a CCSP is theoretically over infinite space. However, due to the computer's limitation to represent real numbers, approximation using *F*-numbers is used. The search starts considering the smallest *F*-box enclosing D. Then a pruning step starts, which tries to remove inconsistent options from the box. Which results in returning a new *F*-box or a union of *F*-boxes.

The pruning step consists of reducing an *F*-box (or a union of *F*-boxes) A to a smaller *F*-box (or a union of *F*-boxes) A'. The pruning algorithm must make sure that there are no potential solutions removed from the original box. Nonetheless, the filtering algorithm may be unable to prune some inconsistencies due to the limited representation power.

The branching step comes after pruning, which consists of dividing the pruned *F*-box into m smaller *F*-boxes, such that, the union of the m *F*-boxes must be the same as the original *F*-box. Then afterwards pruning is applied to the divided boxes, trying to remove further options which are not solutions.

The technique described before is called branch and prune. For the sake of simplification of the domain's representation, most solving strategies impose that only the single F-boxes should be presented (as opposed to a union of *F*-boxes). In that sense, pruning corresponds to narrowing the original *F*-box into a smaller one, where the lengths of some *F*-intervals are decreased by some filtering algorithms (or if it became empty, proving the original *F*-box to be inconsistent). The branching step usually consists of splitting the original *F*-box into two smaller *F*-boxes by splitting one of the original variable domains around an *F*-number, which is in most algorithms the *F*-number representing the mid-value of the *F*-interval of its domain.

The filtering algorithms are used for pruning the variable domains. They are based on constraint propagation techniques. The propagation process is described as a successive pruning to the variables domains'. This is done by applying **narrowing functions** associated to the constraints of the CCSP. Evaluation of the narrowing functions is done by algorithms that take advantage of the techniques of interval analysis.

In the propagation algorithm, a narrowing function is the mapping between elements in a domain: A, and A', such that the new elements in A' are obtained by eliminating some value from A. In the following, the definition of the narrowing function is shown.

Definition 3.12. Let $P = (X, D, C)$ be a CCSP. A **narrowing function** NF associated with a constraint $c = (s, \rho)$ (with $c \in C$) is a mapping between elements of 2^D ($Domain_{NF} \subseteq 2^D$ and $Codomain_N F \subseteq 2^D$) with the following properties (where A is any element of DomainNF):
P$_1$) $NF(A) \subseteq A$ (contractance)
P$_2$) $\forall_{d \in A}\ d \notin NF(A) \rightarrow d[s] \notin \rho$ (correctness)

From property **P1**, it is assured that the new domain $NF(A)$ is smaller than A. Moreover, with property **P2**, the narrowing function does not remove any valid solution for the CCSP.

A narrowing function has to be monotonic and idempotent according to [38] or at least monotonic according to [4].

Definition 3.13. Let $P = (X, D, C)$ be a CCSP. Let NF be a **narrowing function** associated with a constraint C. Let A_1 and A_2 be any two elements of $Domain_{NF}$. NF is respectively monotonic and idempotent iff the following properties hold:
P3) $A_1 \subseteq A_2 \rightarrow NF(A_1) \subseteq NF(A_2)$ (monotonicity)
P4) $NF(NF(A_1)) = NF(A_1)$ (idempotency)

Input: set Q of narrowing functions, a box $B \subseteq D$
Output: a box $B^{'} \subseteq B$

```
1  function prune(Q,B)
2  S ← ∅;
3  while Q ≠ ∅ do
4  |  choose NF ∈ Q ;
5  |  B' ← NF(B);
6  |  if B' = ∅ then
7  |  |  return ∅ ;
8  |  end
9  |  P ← {NF' ∈ S : ∃_{x∈Relevant_{NF'}} B[x] ≠ B'[x]};
10 |  Q ← Q ∪ P;
11 |  S ← S\P;
12 |  if B' = B then
13 |  |  Q ← Q\{NF};
14 |  |  S ← S ∪ {NF};
15 |  end
16 |  B ← B';
17 end
18 return B;
```

FIGURE 3.1: Standard propagation algorithm

The standard propagation algorithm in figure 3.1 takes two arguments: Q, which is the set of all narrowing functions associated with the constraints in the CCSP, and B, which is a subset of the power-set of the domains of the CCSP.

In the beginning of the algorithm, the set S is initialized to the empty set. S contains the narrowing functions for which B is necessarily a fixed-point. Then, until Q is not empty, it starts applying the narrowing functions in Q by selecting a narrowing function NF, and applying it to B to get B'. Then it checks: if the domain of B' is empty, then there is no solution for the problem returning an empty set.

However, if B' is not empty, the set P is defined as a subset of S composed of all the elements of S for which B' is no longer guaranteed to be a fixed-point. These elements,

which are the narrowing functions with relevant variables whose domains were changed by applying NF to B, are moved from S to Q to be applied again on B. Moreover, if B' is a fixed point for B; meaning $B = B'$, then NF is moved to S. Then finally B is updated with the values in B' and the loop repeats again.

Many types of narrowing functions have been developed that follows the monotonicity and idempotency rules. There are many techniques to associate narrowing functions to the constraints of a CCSP.

One important technique is the *constraint decomposition method* [24, 44], mentioned in [8]. It is based on the transformation of complex constraints into an equivalent set of primitive constraints that can be solved with respect to each variable.

First we decompose the set of original constraints into a set of primitive constraints possibly by adding new variables. For example, the complex constraint $(x_2-x_1)^2\times x_1 = 1$ is decomposed into the primitive constraints: $\{x_3 = x_2 - x_1, x_4 = x_3, x_4 \times x_1 = 1\}$ with the addition of the two new variables x_3 and x_4.

The next step of the constraint decomposition method is to solve algebraically each primitive constraint wrt. each variable in the scope and to define an interval function enclosing the respective projection function. This is always possible because the constraints are primitive. However, an extra care must be taken due to the in-definition of some real expressions for particular real-valued combinations.

For example the primitive constraint $x_3 = x_2 - x_1$ yeilds the following narrowing functions for reducing the domains of each variable:

- $NF_1(I_1) \rightarrow (I_2 - I_3) \cap I_1$.
- $NF_2(I_2) \rightarrow (I_3 + I_1) \cap I_2$.
- $NF_3(I_3) \rightarrow (I_2 - I_1) \cap I_3$.

There are several other narrowing functions like the Newton Method [3], and several modifications to the decomposition methods [24] and the Newton methods [46].

Despite being a finite search space, the domains of a CCSP usually contain a huge number of elements, and any strategy to navigate over it must be aware that the underlying real-valued search space is infinite. To be effective, a solving strategy cannot rely exclusively on branching, expecting the splitting process to stop eventually because the search space is finite. In fact, the splitting process is theoretically guaranteed to stop but the combinatorial number of necessary splits usually prevents such stopping from being achieved in a reasonable amount of time. One approach often adopted imposes conditions on the branching process, for instance, branching may only be performed on lattice elements with some variable domains larger than a predefined threshold, and this may only be done by splitting one of these domains.

3.3 Consistency Techniques

The fixed-points of a set of narrowing functions associated with a constraint c characterize a local property enforced among the variables $x_1, x_2, \ldots, x_n$ of the constraint scope. Such property is called *local consistency*. It mainly depends on these narrowing functions which are associated with only one constraint. Moreover, it defines the value combinations that are not removed from the variables. Local consistency is a partial consistency, which means that when it is imposed on a CCSP problem, it does not remove all the inconsistent combinations between its variables.

The local consistencies that are in use for the CCSP is a modification of the arc-consistency which is a local consistency mainly used for CSP over finite domains and was introduced in [30]. Arc-consistent constraints are the constraints for which each value in the variables involved in the constraint has a consistent value with the other variables' values of the constraints. The definition of arc-consistency is as follows.

Definition 3.14. Let $P = (X, D, C)$ be a CSP. Let $c = (s, \rho)$ be a constraint of the CSP. Let A be an element of the power set of D ($A \in 2^D$). The constraint c is arc-consistent wrt A iff:
$\forall_{x_i \in s}\ \forall_{d_i \in A[x_i]}\ \exists_{d \in A[s]}\ (d[x_i] = d_i \wedge d \in \rho)$

Arc consistency can not be enforced with CCSP since the domains of the variables in the problem are infinite, and the machine has a memory limitation. Therefore, the real values are approximated into canonical *F*-interval. The best approximation of arc-consistency wrt a set of real-valued combinations is the set approximation of each variable domain.

This is the idea of the *interval-consistency* [24, 44]. A constraint is interval-consistent wrt a set of value combinations iff for each canonical *F*-interval representing a variable sub-domain there is a value combination satisfying the constraint. Interval-consistency has the following formal definition.

Definition 3.15. Let $P = (X, D, C)$ be a CCSP. Let $c = (s, \rho)$ be a constraint of the CCSP ($c \in C$). Let A be an element of the power set of D ($A \in 2^D$). The constraint c is interval-consistent wrt A iff: $\forall_{x_i \in s}\ \forall_{[a..a^+] \subseteq A[x_i]}\ \exists_{d \in A[s]}\ (d[x_i] \in (a..a^+) \wedge d \in \rho)$
$\wedge\ \forall_{[a] \subseteq A[x_i]}\ \exists d \in A[s]\ (d[x_i] \in (a^-..a^+) \wedge d \in \rho)$ (where a is an *F*-number).

Interval-consistency can only be enforced on primitive constraints where the set approximation of the projection function can be obtained using interval arithmetic. Moreover, according to [24], in practice, the enforcement of interval-consistency can be applied only to small problems.

Several other kinds of local consistencies were developed. For example, the *Hull-consistency* [28], and the *Box-consistency* [3] that are variants of arc-consistency enforced only on the bounds of each variable.

In addition to local consistency, *Higher Order Consistency* provides better pruning to the variable domains. The main idea of higher order consistency is to have some global

view on a subset of the constraints in the CCSP or the even the whole set C. Several higher order consistency algorithms were introduced, including *3B-consistency* [28], *Bound consistency* [40], and *KB-consistency* [28].

4

Hybrid Local Search Constraint Optimization Algorithms

After introducing continuous optimization in chapter 2 and continuous constraint satisfaction in chapter 3, we introduce our main work, in which we combine techniques of continuous optimization problems, whether constrained or unconstrained, with the branch-and-bound algorithms of CCSP.

In recent years, there has been a lot of research done in the field of hybrid constraint programming and hybrid constraints with local search [43]. This research is focusing mainly on constraints over finite domains [13]. There are many frameworks developed for constraint programming with the option of using local search techniques to satisfy the problem. One of the most famous frameworks which is mainly developed for CSP over finite domains is Comet [20].

In this thesis, we discuss the combination of branch-and-bound technique that is used to propagate constraints and solve CCSP, with continuous local search over the branched and pruned boxes of the CCSP. When solving an optimization problem, trying to get the global minimum, constraint propagation techniques are used to prune the domains of the variables in the problem after branching the search space into several boxes. We use continuous local search techniques over every box, trying to get a local minimum within that box. By comparing the results obtained for the local minima of each box, we can keep track of the minimal "local minimum", which is the local minimum which has the lowest value found. Therefore, we get an estimate of the global minimum of the problem over the search space explored.

Figure 4.1 describes the general algorithm used for the local search with CCSP. The

algorithm takes a CCSP with an optimization function f. In other words, it takes a constrained optimization problem with the function f that will be minimized, and a set C of constraints that are transformed into the form of constraints, appearing in the standard optimization problem in equation 2.1.

In line 1, $\mathcal{B}$ is a set of boxes which is initialized with the initial box for the branch and prune algorithm. Afterwards, in lines 2 and 3 we initialize the variables where the minimal objective function value and the minimal local point found will be stored.[1] A loop starts in line 4 until $\mathcal{B}$ is empty. In line 5, a box b is selected from $\mathcal{B}$ according to a selection criterion. This criterion selects b with the highest potential of finding a better local minimum. Branching is applied on the selected box b, creating a set of boxes $\{b_1, \ldots, b_n\}$. Then another loop starts for every box b_i, starting by pruning the box b_i in line 8.

Input: A continuous optimization Problem P with the function f (which can also be a CCSP with an objective function)
Output: a point x, the global minimum found so far.

```
1  B ← {(Initial box b)};
2  minimalPoint ← NULL;
3  minimalValue ← ∞;
4  while B is not empty do
5      b ← B.takeBox();
6      {b_1, ..., b_n} = b.branch();;
7      foreach b_i ∈ {b_1, ..., b_n} do
8          b_i.prune();
9          if b_i is not empty then
10             <succeed, point> ← applyLocalSearch(b_i);
11             if succeed == true then
12                 value ← f(point);
13                 if boxValue < minimalValue then
14                     minimalValue ← value;
15                     minimalPoint ← point;
16                 end
17             end
18             if stopping criteria reached then
19                 return minimalPoint;
20             end
21             add b_i to B ;
22         end
23     end
24 end
25 return minimalPoint;
```

FIGURE 4.1: Pseudo code of the general CCSP-LS hybrid technique

Afterwards, an emptiness check is applied on b_i in line 9. If b_i is not empty, the local

[1] $MinimalPoint$ and $MinimalValue$ are global variables that can be accessed from any function.

search optimization will be applied within the box, taking into consideration that some of the local search algorithms we discuss will always maintain the bounds of the box. Others have the option of trying to get an even better optimal point by going out of the bounds of the box b_i.

The `applyLocalSearch`(b_i) algorithm returns the pair $<succeed, point>$. The first parameter in the pair is a boolean variable which is *true* if the algorithm succeeded to find a local minimum in the b_i, and *false* otherwise. If $succeed$ is *true*, then the box local minimum is saved in $point$, and its objective function value is calculated and saved in $value$. Then the $value$ is checked if it has the minimal value of f so far. If that is the case, $minimalValue$ and $minimalPont$ are updated as it is shown in lines 14 and 15.

At the end of every iteration, a stopping criterion is checked. Many criteria can be used, such as the difference between the lower bound of the objective function interval evaluated by interval arithmetics to the $minimaValue$. If the stopping criteria is not reached, b_i is added to $\mathcal{B}$.

We will use *Procure* [42], which is a probabilistic continuous domain constraint framework, for the implementation for our algorithms and obtaining the results. We will discuss in brief how *Procure* works in section 4.1.

In section 4.2, we implemented a random search algorithm for the sake of comparison with the local search algorithms. In later sections, sec. 4.3 and 4.4, we will discuss two local search algorithms we implemented with the logic that supports it. These two algorithms depend on the concept of the steepest descent local search. However, they have different step length calculation and penalty functions.

4.1 Procure, a quick introduction

All the algorithms we propose and investigate in this work are implemented using Procure. Procure is a probabilistic continuous domain constraint framework which is built over *RealPaver*[18], which is an interval-based solver. Procure is developed by the Centria group at the new University of Lisbon. Using the `C++` programming language and following an object oriented design, this solver provides a set of useful continuous constraint methods, and its design makes it easily extensible.

Procure provides many mathematical methods for calculation, such as calculating the derivative of a function and partial derivatives. It uses branch-and-bound algorithms for constraint propagation. Moreover, with the modularity provided with it, the simplicity of changing the search algorithm within the branched boxes comes in handy.

It implements the general CCSP-LS algorithm that was shown in figure 4.1. However, it does not use the local search algorithm over every processed (branched and pruned) box. What it simply does, is that it takes the *mid-points* over all intervals of the variables in the branched box and return them as the point.

In the Procure implementation, $\mathcal{B}$ is an ordered set of boxes with respect to the box lower Bound of the objective function f. Procure uses interval arithmetic to calculate an

```
1 vector<Procure::Var> x(1);
2  Problem prob(x,{{-3,20}},{x[0] >= 0 },20 + x[0] * sin(x[0]) );
```

FIGURE 4.2: Example 1, simple optimization problem

interval disclosure of f for each box in $\mathcal{B}$ and maintain the ordering of $\mathcal{B}$ ascendingly with respect to the boxes' lower bounds.

Moreover, when a problem is passed to Procure to find the global minimum, an implicit constraint c_{obj}, shown in equation 4.1, is added to the set of constraints C of the optimization problem. c_{obj} makes sure that the new local minimum found by the algorithm is having a lower objective value than the current local minimum. c_{obj} also helps in pruning the set of boxes $\mathcal{B}$. From this set we remove the boxes that do not have any potential of finding the global minimum, meaning the boxes having a lower bound with a higher value than the local minimum found so far. Assume the set $\mathcal{B}$ has the boxes $b_1, \ldots, b_n$. c_{obj} imposes that the objective function value of the lower bound (LB) of b_i is less than or equal to the objective function value of the minimal point ($minimalValue$). c_{obj} is updated whenever the $minimalValue$ is changed.

$$c_{obj} = f(x) \leq minimalValue. \tag{4.1}$$

In order to solve an optimization problem, in addition to the optimization function f, we specify the variables X, the initial domains D and the constraints C. Figure 4.2 shows an example of an optimization function which is written in Procure syntax. This example will be used throughout this chapter to make comparisons with different search algorithms.

Figure 4.2 shows an optimization problem with one variable `x[0]`, for simplicity, let's call it x. x has the initial domain of $[-3, 20]$ with one constraint in the system, that is $x \geq 0$. The optimization function is $20 + x \times sin(x)$, that we want to minimize. In figure 4.3, the minimization function is presented. There are four local minima in the feasible area of the problem:

- $x = 0$ with $f(x) = 20$.
- $x \approx 4.99$ with $f(x) = 15.185$.
- $x \approx 11.0855$ with $f(x) = 8.95929$.
- $x \approx 17.3364$ with $f(x) = 2.6924$, which is also the global minimum.

Solving this optimization problem with Procure's *mid-point* function needed to use the branch and prune algorithms 4 times to obtain 4 pruned boxes. The final result for the minimal point found is at 2.692. The time taken for the algorithm is 0.07 second. This is due to the simplicity of the objective function. In the next chapter, we will see the impact of local search on a bigger scale example.

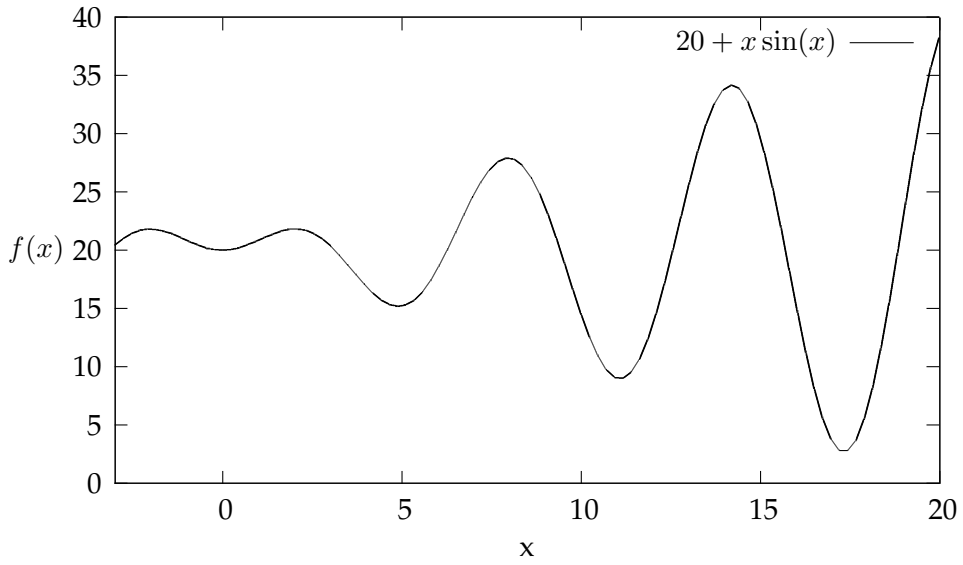

FIGURE 4.3: Plot of Procure example

4.2 Random Local Search

In this local search algorithm, we use a completely random selection algorithm for obtaining a point x from a box b. Simply by taking the current box b as an input, we randomly choose x, and then check whether it is in the feasible area of the problem. If x is feasible, the search terminates, and x is returned. On the other hand, if this is not the case, then a new random point is obtained. This process will continue until the maximum number of $restarts$ is reached. If the restarts number is reached, then the algorithm fails to find any feasible point in its random search and terminates.

The local search algorithm which appears in figure 4.4 does what is mentioned earlier, and it returns the pair $<succeed, point>$. If the algorithm succeeded to find a feasible point x, then $succeed$ is set to *true* and the $point$ takes the obtained point. Otherwise, it returns <*false*,NULL>.

With the random local search, the time of solving an optimization problem is varied over a long time. Therefore, when we get to test this algorithm, we take an average of five runs of the problem and get the average timing.

In the problem that appeared in figure 4.2. We ran the problem five times using the local search algorithm. It took an average of 0.07 seconds which is the same time of the mid-point algorithm. Most of the search space is in the feasible area. This results in most of the selected random points being feasible. This allows the branching to go faster, removing all the boxes that do not satisfy the global constraint c_{obj}. Moreover, the number of boxes branched and checked are on average 13 boxes, which is considered to be many compared to the simplicity of the optimization problem.

Input: a box b
Output: A pair $<Bool, Point>$

```
1 restarts ← 50;
2 for i ← 1 to restarts do
3 |   x ← random point in b;
4 |   if x is feasible then
5 |   |   return <true,x>;
6 |   end
7 end
8 return (false,NULL);
```

FIGURE 4.4: Random Local Search algorithm

4.3 Armijo Rule with Quadratic Penalty Steepest Descent

In this local search algorithm, we use the steepest descent method for obtaining a new point x_{k+1} from x_k. Steepest descent algorithm is mentioned in section 2.9.

There are two main options for implementing this algorithm: how to represent the constraints and how to select the step length. Constraints are presented in the problem using the quadratic penalty method. In other words, the count of the penalty caused by violating the constraints in the system is added to the value of the overall objective function. This is done by changing the objective function f to Q, where Q takes into account the number and amount of violations in every constraint. The technique of quadratic penalty function is discussed in 2.1.2, equation 2.14.

The selection of the step length α_k with respect to the direction d_x is determined by the *Armijo rule*, as shown in equation 2.8. The algorithm's stopping criteria mainly depends on the iterative settings of the algorithm. A counter is set to a number m, then by the end of every iteration the boolean value $Q(x_{k+1}) \geq Q(x_k)$ is checked. The counter decreases whenever this check is true, and resets when it is false. If for m consecutive iterations the check succeeded, the algorithm terminates, returning x_k.

The main idea of this algorithm is that when the violated constraints are added as a penalty in the original objective function f to obtain Q, then using the steepest descent in the decreasing direction of Q, the penalty value of x_{k+1} is going to be less than that of x_k.

Figure 4.5 shows the algorithm with the stopping counter *stops* mentioned earlier. The input is a box b with a vector x of n variables. The algorithm tries to obtain a local minimum inside b with respect to the newly obtained objective function Q. It returns a pair $<succeed, point>$. $succeed$ is *true* if the algorithm found a local minimum, and it sets $point$ to this local minimum point. It returns *false* if it did not succeed in finding a local minimum.

The algorithm starts by setting the restarts number, which in this case is 50. Then with every restart, x_k is set to be a random point in b. A stopping flag which is set to 5 acts as a stopping criteria when the local search starts. In case it could not find a better point in

the next five runs, the search stops. The variable μ is set to 1.

Afterwards, in every iteration of the local search, a new objective function Q is calculated using the function `getQuadPenaltyFunction`(f,x_k,μ) which will be discussed later. In addition to calculating Q, the step ratios are calculated for every variable in b. This is done using the function `getArmijoSteps`(Q,x_k) that returns an array of ratios; a ratio for each variable.

Input: a box b
Output: A pair $<Bool, Point>$
Result: Getting a local minimum in a box b

```
1  restarts ← 50;
2  for i ← 1 to restarts do
3      x_k ← random point in b;
4      stop ← 5;
5      μ ← 1;
6      while stop ≥ 0 do
7          Q ← getQuadPenaltyFunction(f,x_k,μ) ;
8          Steps ← getArmijoSteps(Q,x_k) ;
9          for j ← 1 to n do /*n is number of variables in x_k*/
10             x_(k+1).j ← x_k.j − Steps[j] × ∇Q_j ;
11         end
12         Q' ← getQuadPenaltyFunction(f,x_(k+1),μ) ;
13         if Q'(x_(k+1)) ≤ Q(x_k) then
14             \ * Modification Here * \
15             x_k ← x_(k+1);
16             stop ← 5;
17             μ ← 1;
18         else
19             stop ← stop − 1;
20             μ ← μ × 2;
21         end
22     end
23     if x_k is feasible then
24         return <true,x_k>;
25     end
26 end
27 return <false,NULL>;
```

FIGURE 4.5: Steepest Descent with Quad. Penalty and Armijo

In line *9*, a loop starts over the variables in x_k. The new value of each variable in $x_{(k+1).j}$ is obtained by subtracting the corresponding ratio of the variable $steps[j]$ multiplied by the partial derivative of the new function Q with respect to the same variable, from the old value $x_{k.j}$.

A new objective function $Q^{'}$ is obtained in line *12*. It is the penalty objective function which corresponds to the new point x_{k+1}. Note that the number of violated constraints

might change, especially concerning the constraints of the form $c \leq 0$.

Afterwards in line *13*, we check if $Q^{'}(x_{k+1}) \leq Q(x_k)$ holds in order to update the value of x_k to be x_{k+1} and reset the stop counter and μ. However, if this condition is not satisfied, the algorithm will be rerun, but after multiplying μ by 2, the decrease of weight of the penalties in the newly calculated Q. This increases the potential of x_{k+1} to have lower Q in the next iteration. Moreover, the stop counter is decreased. The comment in line 14 will be discussed later.

Furthermore, at the end of every restart, a check for feasibility is done over the obtained final x_k, if it is feasible, then it is a valid local minimum, and the algorithm returns the pair $(true, x_k)$. If it is not, another restart loop commences.

μ is the variable controlling the weight of the penalties in the obtained objective function Q. The function `getQuadPenaltyFunction` is called on every iteration in order to mainly update the $c_i \leq 0$ constraints, so if they are less than or equal to zero, we do not add them in order to maintain the equation 2.14, specifically the part of $\sum_{i=k+1}^{i=m} \max(c_i(x), 0)^2$.

The `getQuadPenaltyFunction` is in figure 4.6. The algorithm is direct, as it performs exactly what the quadratic penalty equation presents. It calculates the new function Q from f by adding all the terms $\frac{1}{2\mu}c^2(x)$ to f in case of c being an equality constraint. However, for inequality constraints, it first checks if the constraint is violated in order to add the term representing this constraint; $\frac{1}{2\mu}c^2(x)$. If it is not violated by x_k, then it is not added to Q.

Input: Objective Function f, a point x, a scalar value μ
Output: Quadratic penalty Objective function Q

```
1  Function getQuadPenaltyFunction(f, x, μ)
2      Q ← f ;
3      foreach constraint c ∈ C do
4          if type of c equals "c_i = 0" then
5              Q ← Q + (1/2μ) c²(x) ;
6          end
7          if type of c equals "c_i ≤ 0" and c(x) > 0 then
8              Q ← Q + (1/2μ) c²(x) ;
9          end
10     end
11     return Q ;
```

FIGURE 4.6: Quadratic Penalty objective function algorithm

Then for selecting the step length in every iteration, we call `getArmijoSteps`. Moreover, we take a step for every variable in b, so for every variable we have its specific step length.

The function `getArmijoSteps`(f, x_k) of figure 4.7 starts by setting the two main variables ρ and c to 0.7 and 10^{-4}, respectively. These values are set according to [50], as these are the best values known in practice for these two variables. The main intuition

of the Armijo function is to backtrack the value of α starting from 1, until it reaches an acceptable area. An acceptable area is an area which the Armijo inequality equation 2.8 is satisfied. When the Armijo condition is satisfied, it guarantees an acceptable decrease in $Q(x_{k+1})$.

The algorithm retries a smaller value of step α over each iteration until the Armijo condition is satisfied. The algorithm does that for each variable in the point vector x. It returns an array of steps, one step for each variable.

Input: An objective function f, a point vector x
Output: Quadratic penalty Objective function Q

```
1  Function getArmijoSteps(f, x)
2      ρ ← 0.7 ;
3      c_1 ← 0.0001 ;
4      Ratios ← new array of the size of x;
5      for k = 0; k ≤ x.size ; k++ do
6          α_k ← 1;
7          repeat
8              value_1 ← f(x − α_k ∇f_k) ;
9              value_2 ← f(x) + c_1 α_k ∇f_k^T ∇f_k ;
10             α_k ← α_k × ρ;
11         until value_1 ≤ value_2;
12         Ratios[k] ← α_k/ρ;
13     end
14     return Ratios ;
```

FIGURE 4.7: Armijo rule line search algorithm

We discuss how this local search algorithm works over the example given in figure 4.2. It starts by choosing x_k randomly, then iterates over it, obtaining better values for the penalty function Q at every iteration. In table 4.8, the results for the iterates are shown, showing the iteration number, the start point of the iteration (x_k), the quadratic function Q, the step ratio α, the value $Q(x_k)$, the obtained x_{k+1}, $Q^{'}(x_{k+1})$, and finally the check if $Q^{'}(x_{k+1}) < Q(x_k)$ are shown.

The value of Q decreases over iterating, converging into the nearest local minimum from the starting point. In the first iteration, x_k is set to be 10.76, which does not violate the single constraint in the system $x \geq 0$, having $Q(x_k) = 9.51858$. By applying the steepest descent for x, which in this case equals $(10.76) - 0.16807((10.76)cos((10.76)) + sin((10.76))) = 11.3418 = x_{k+1}$. It has also a better value for the objective function $Q^{'}(x_{k+1}) = 9.33135$.

The objective function value converges slowly as the algorithm iterates more into the closest local minimum of the box, which is in this case $x_k = 11.0856$ with $f(x_k) = 8.96$. After the tenth iteration, the convergence to the local minimum becomes slower such that it requires 138 iterations to finally reach the optimal value of the local minimum with almost unnoticeable change. In practice, this is a huge number of unneeded iterations to

It	x_k	Q	α	$Q(x_k)$	x_{k+1}	$Q^{'}(x_{k+1})$	$<?$
1	10.7654	$20 + x * sin(x)$	0.16807	9.51858	11.3418	9.33135	t
2	11.3418	$20 + x * sin(x)$	0.16807	9.33135	10.853	9.25717	t
3	10.853	$20 + x * sin(x)$	0.16807	9.25717	11.2786	9.17004	t
4	11.2786	$20 + x * sin(x)$	0.16807	9.17004	10.9106	9.12867	t
5	10.9106	$20 + x * sin(x)$	0.16807	9.12867	11.2337	9.08322	t
6	11.2337	$20 + x * sin(x)$	0.16807	9.08322	10.9517	9.05882	t
7	10.9517	$20 + x * sin(x)$	0.16807	9.05882	11.2003	9.03364	t
8	11.2003	$20 + x * sin(x)$	0.16807	9.03364	10.9821	9.01886	t
9	10.9821	$20 + x * sin(x)$	0.16807	9.01886	11.175	9.00439	t
10	11.175	$20 + x * sin(x)$	0.16807	9.00439	11.0052	8.99532	t
20	11.112	$20 + x * sin(x)$	0.16807	8.96321	11.062	8.96239	t
30	11.0934	$20 + x * sin(x)$	0.16807	8.95964	11.0786	8.95956	t
40	11.0879	$20 + x * sin(x)$	0.16807	8.95932	11.0835	8.95932	t
50	11.0862	$20 + x * sin(x)$	0.16807	8.95929	11.0849	8.95929	t
60	11.0857	$20 + x * sin(x)$	0.16807	8.95929	11.0854	8.95929	t
70	11.0856	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
80	11.0856	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
90	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
100	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
110	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
120	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
133	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	t
134	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	f
135	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	f
136	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	f
137	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	f
138	11.0855	$20 + x * sin(x)$	0.16807	8.95929	11.0855	8.95929	f

FIGURE 4.8: Steepest Descent with Quad. Penalty and Armijo steps Iterations for one box

reach the optimal local minimum.

Therefore, it took almost 0.7 seconds to solve this problem, due to the very small ratio of convergence to the local minimum. This is the main reason we introduce the *Modified Armijo Rule with Quadratic Penalty Steepest Descent* algorithm. The modification, which we show in 4.9, is inserted into the algorithm; figure 4.5, instead of line 14 in figure 4.5.

What that change does is that instead of resetting the $stop$ variable to 5 iterations, it decrements it by 1 if the new value $Q^{'}(x_{k+1})$ is 95% close to the old value $Q(x_k)$ or more. By using this new technique, the chances of eliminating unneeded iterations with very small convergence rate increases.

Table 4.10 supports this claim. This eliminates a huge amount of unneeded iteration in practice. We use the same box, with the same start point. The algorithm stops after only five iterations, giving a close result to the original algorithm.

Using the modified version of the Armijo Rule with Quadratic Penalty Steepest Descent algorithm enhanced the execution time dramatically, giving 0.03 seconds to solve

```
if Q'(x_{k+1})/Q(x_k) ≥ 0.95 then
    x_k ← x_{k+1};
    stop ← stop − 1;
    μ ← 1;
    Continue;
end
```

FIGURE 4.9: Modification of Steepest Descent with Quad. Penalty and Armijo

It	x_k	Q	α	$Q(x_k)$	x_{k+1}	$Q^{'}(x_{k+1})$	$<?$
1	10.7654	$20 + x * sin(x)$	0.16807	9.51852	11.3418	9.33132	t
2	11.3418	$20 + x * sin(x)$	0.16807	9.33132	10.853	9.25714	t
3	10.853	$20 + x * sin(x)$	0.16807	9.25714	11.2786	9.17002	t
4	11.2786	$20 + x * sin(x)$	0.16807	9.17002	10.9107	9.12866	t
5	10.9107	$20 + x * sin(x)$	0.16807	9.12866	11.2337	9.08321	t

FIGURE 4.10: **Modified** Steepest Descent with Quad. Penalty and Armijo steps iterations for one box

this problem, instead of 0.7 seconds.

To show an example of the penalty function in action, in table 4.11, we choose the starting point x_k in the box from $[-3, 1]$ to be -1, such that it violates the constraint $x \geq 0$, the penalty function is applied and the algorithm converges x_k to the feasible area, obtaining a feasible solution which is close to the local minimum of this box: $x = 0$.

In conclusion, the Armijo Rule with Quadratic Penalty Steepest Descent algorithm converges very rapidly in the beginning for the nearest local minimum in the box b. It depends on the quadratic penalty function, which adds the penalties of the violated constraints into the overall value of the objective function. In some cases, if the violations are too many, it might need a longer time to converge into a feasible area. Moreover, the calculation of the step function α is accurate as it almost guarantees a decrease in the objective function. However, the cost of reaching α is high as it might take several iterations to obtain it.

It	x_k	Q	α	$Q(x_k)$	x_{k+1}	$Q^{'}(x_{k+1})$	$<?$
1	-1	$20+x*sin(x)+$ $0.5*x^2*\mu^{-1}$	0.343	22.8415	0.845948	20.6333	t
2	0.845948	$20 + x * sin(x)$	1	20.6333	-0.463534	20.637	f
3	0.845948	$20 + x * sin(x)$	1	20.6333	-0.463534	20.4221	t
4	-0.463534	$20+x*sin(x)+$ $0.5*x^2*\mu^{-1}$	0.2401	20.637	0.188546	20.0353	t
5	0.188546	$20 + x * sin(x)$	1	20.0353	-0.18409	20.1015	f
6	0.188546	$20 + x * sin(x)$	1	20.0353	-0.18409	20.0676	f

FIGURE 4.11: **Modified** Steepest Descent with Quad. Penalty and Armijo steps Iterations with constraints violations

The algorithm takes a box b as an input. However, it is not bound to that box when it searches for a feasible local minimum. Therefore, in this case, the points inside b act as potential starting points for the random assignment. The final solution might be a point from outside the box, but inside the overall domain D of the variables[2] .

4.4 Box Ratio with Separate Penalty Steepest Descent

In this section, we discuss another local search algorithm, which is also based on the steepest descent algorithm. However, it differs in handling the constraints in the CCSP. Moreover, the calculation of the step length depends on the box size.

The box ratio with separate penalty steepest descent algorithm presents only violated constraints by using the decreasing direction of the violated constraints gradient. Therefore, with a modification to the steepest descent equation 2.9, we get a steepest descent equation that accounts for the constraint's gradient. The new steepest descent equation is as follows:

$$f(x_{k+1}) = f(x_k) - \mathbf{r}\alpha_k \nabla f_k - \sum_{i=0}^{v(C,x_k).size} \textbf{(1-r)}\alpha_k \nabla v(C,x_k)_i \tag{4.2}$$

$v(C, x_k)$ is a function that takes the set of constraints C and a point x_k and returns an array of the violated constraints in C with respect to x_k. $0 \leq \mathbf{r} \leq 1$ is a weight constant that divides the weights of the effect of the gradient of the objective function f or the violated constraints $v(c, x_k)$. $\nabla v(C, x_k)_i$ is the gradient of the constraint in position i of the array $v(C, x_k)$.

Moreover, the ratio α is depending on the width of the box b and the direction of the gradient ∇f.

Definition 4.1. A box ratio is the ratio between the distance of a variable x and the lower bound of the box b with respect to x; b_x, to the width of b_x in case $\nabla f(x) > 0$, or the distance between x and the upper bound of b in case $\nabla f(x) < 0$. It is 0 otherwise, since $\nabla f_x = 0$ means that x is an optima.

$$boxRatio(f, x, b) = \begin{cases} \frac{x - b_x.lb}{b_x.ub - b_x.lb} & \nabla f(x) > 0 \\ \frac{b_x.ub - x}{b_x.ub - b_x.lb} & \nabla f(x) < 0 \\ 0, \quad otherwise \end{cases} \tag{4.3}$$

$b_x.lb$ and $b_x.ub$ are the lower bound and the upper bound of b_x respectively. Box ratio is a greedy step length calculation as it calculates the distance needed to reach the bound of the box from the point x. Then box ratio is used to determine several steps on the gradient of f.

The stopping criteria of the algorithm depends on the step lengths taken by the line search algorithm which is this case the box ratio algorithm. A counter is set to a number

[2]The algorithm is implemented such that it does not allow the variables' value to be outside the domain

m, by the end of every iteration x_{k+1} is checked if it is *better* than x_k according to figure 4.14, that is discussed later. If it is not the case, then m is decremented and the algorithm terminates when the counter reaches 0.

The main idea in this algorithm is to separate the penalty calculation from the objective function f and to use a step length that depends on the length of the box.

In figure 4.12, we show the algorithm of box ratio with separate penalty steepest descent. It has the same input, which is a box b, as the other local search algorithms. Returning $pair(Bool, Point)$ with the same semantics as mentioned in algorithm 4.5. The restart number in this algorithm is also 50.

With every restart, the point x_k takes a random value from the box b. A counter $stop$ is intialized to 5. For calculating the ratio, the value stored in x_k has to be changed. Therefore, $changedFlag$ is set to *true* with every restart.

In line 6, **r** is set to 0.25, meaning that the weight of the change for the direction of ∇f is 0.25, compared to 0.75 for the negative direction of ∇c. We use this ratio because it is more important to find a feasible solution than finding a point with a small value of f that is not feasible.

In line 9, if $changedFlag$ is true, the array $ratios$ takes the returned value of the function `getBoxRatio`, which will be discussed later. $halfRatios$ is half the value of the values saved in $ratios$ array such that for every iteration there are 2 step lengths that will be compared.

Line 14 starts a loop over every variable in the box b, applying the modified steepest equation that includes the gradient of the violated constraints in the equation 4.2. This is done over the two points x'_{k+1} and x''_{k+1}, using the $ratios$ and $halfRatios$, respectively.

In line 25, the function `isBetterPoint` checks which point x'_{k+1} or x''_{k+1} is better with respect to the objective function value and the penalty value. The function is discussed later in figure 4.14. x_{k+1} takes the better value.

x_{k+1} is then checked with x_k using the same function `isBetterPoint`. If x_{k+1} is a better point, then x_k takes the value of x_{k+1} and the $changedFlag$ set to *true* as the value of x_k changed and the $stop$ counter is reset to 5. If it is not the case that x_{k+1} is a better point, then the stopping criteria variable $stop$ is decremented and the values in $ratios$ are halved and the same for $halfRatios$.

x_k is checked for feasibility. If it is feasible, then the pair (*true*,x_k) is returned. If not, the loop restarts. The modification in line 23 is discussed later.

The overall intuition of this algorithm is to try to obtain a new point x_{k+1} by going through a logarithmic search over the gradient line of f and the gradient line of the violated constraints. This occurs until a point x_{k+1}, which is better than x_k, is eventually found, or the algorithm terminates.

The function `getBoxRatio` is a direct application of the box ratio equation 4.3. In figure 4.13, the box ratio function is mentioned. $b_j.lb$ is the lower bound of b with respect to the variable in the j^{th} position of b. $b_j.ub$ is the upper bound of the same variable.

The algorithm loops over every variable in b, obtaining the box ratio according to

Input: a box b
Output: A pair $(Bool, Point)$
Result: Getting a local minimum in a box b

```
1  restarts ← 50;
2  for i ← 1 to restarts do
3      x_k ← random point in b ;
4      stop ← 5 ;
5      changedFlag ← true;
6      r ← 0.25;
7      while stop ≥ 0 do
8          if changedFlag then
9              ratios ← getBoxRatio(b,x_k,f) ;
10             for j = 0; j ≤ ratios.size ; j++ do
11                 halfRatios[j] = 1/2 × ratios[j];
12             end
13         end
14         for j ← 1 to n do  /*n is number of variables in x_k*/
15             x'_(k+1).j ← x_k.j − (r × ratios[j] × ∇f_j);
16             x''_(k+1).j ← x_k.j − (r × halfRatios[j] × ∇f_j);
17             foreach constraint c ∈ C do
18                 if isViolated(c,x_k) then
19                     x'_(k+1).j ← x'_(k+1).j − (1-r) × ratios[j] × ∇c_j ;
20                     x''_(k+1).j ← x'_(k+1).j − (1-r) × halfRatios[j] × ∇c_j ;
21                 end
22             end
23             \ * Modification Here * \
24         end
25         if isBetterPoint(x'_(k+1),x''_(k+1)) then
26             x_(k+1) ← x'_(k+1);
27         else
28             x_(k+1) ← x''_(k+1);
29         end
30         if isBetterPoint(x_(k+1),x_k) then
31             x_k ← x_(k+1);
32             stop ← 5;
33             changedFlag ← true;
34         else
35             stops ← stops − 1;
36             changedFlag ← false;
37             for j = 0; j ≤ ratios.size ; j++ do
38                 ratios[j] = halfRatios[j];
39                 halfRatios[j] = 1/2 × ratios[j];
40             end
41         end
42     end
43     if x_k is feasible then
44         return (true,x_k);
45     end
46 end
47 return (false,NULL);
```

FIGURE 4.12: Steepest Descent with Separate penalty and Box ratio

equation 4.3.

The comparison between two points in this local search algorithm is performed using `isBetterPoint`. The function takes two points x_1 and x_2, and returns a boolean value, which is *true* if it has a lower violations (penalty) value of x_1. The penalty of a point x is calculated with the function `getPenalty`. If both penalties are the same, then $f(x_1)$ and $f(x_2)$ are checked. If $f(x_1)$ has a lower value, then it returns *false*. The function returns *false* otherwise, meaning that x_1 is better than x_2.

Input: Box b, objective Function f and a point x
Output: Array $ratios$

```
1  Function getBoxRatio(x, f)
2      for j = 0; j ≤x.size;j++do
3          gradientValue ← ∇f_{x_j};
4          boxLength_j ← mathitb_j.ub − mathitb_j.lb;
5          if gradientValue < 0 then
6              ratios[j] ← (x_j − b_j.lb) / boxLength_j ;
7          else
8              if gradientValue > 0 then
9                  ratios[j] ← (b_j.ub − x_j) / boxLength_j ;
10             else
11                 ratios[j] ← 0 ;
12             end
13         end
14     end
15     return ratios ;
```

FIGURE 4.13: Box ratio algorithm

The `isBetterPoint`(x_1,x_2) function is shown in figure 4.14. It calculates the penalties of the two points using the function `getPenalty`. It gives a priority to the penalty value over the value of the objective function f. Therefore, it checks first if x_1's penalty value is less than the penalty value of x_2, in which case it returns *true*. If both penalty values are the same, then the value of f is checked, returning true if $f(x_1) < f(x_2)$, and *false* otherwise.

The function `getPenalty` calculates the penalty of a point x with respect to the constraints set C given in the problem. We are assuming that C is global and accessible from any function in the program.

Figure 4.15 shows two algorithms. The first one is `getPenalty`, which calculates the penalty value of a point. In addition, the function `isViolated` checks if a single constraint c is violated with respect to a point x.

`isViolated` checks the violation of a single constraint c with respect to the point x. The violation is checked according to the type of the constraint. The valid types of constraints are the constraints that appear in equation 2.1. The evaluation of the expression

Input: Points x_1 and x_2
Output: Boolean *true* or *false*

```
1  Function isBetterPoint(x1, x2)
2      penalty1 ← getPenalty(x1);
3      penalty2 ← getPenalty(x2);
4      if penalty1 < penalty2 then
5          return true
6      else
7          if penalty1 = penalty2 then
8              return (f(x1) < f(x2) );
9          end
10     end
11     return false;
```

FIGURE 4.14: Better Point algorithm

Input: Point x
Output: real value

```
1  Function getPenalty(x)
2      penalty ← 0;
3      foreach constraint c ∈ C do
4          if isViolated(c,x) then
5              penalty ← penalty + c(x)^2;
6          end
7      end
8      return penalty;
```

Input: Constraint c, Point x
Output: boolean value

```
9  Function isViolated(c,x)
10     if type of c equals "ci = 0" then
11         return c(x).ub < 0 || c(x).lb > 0;
12     end
13     if type of c equals "ci ≤ 0" then
14         return c(x).lb > 0
15     end
```

FIGURE 4.15: Penalty and Violation related functions

It	x_k	$f(x_k)$	$x_k.pen$	rat	$\frac{1}{2}rat$	x'_{k+1}	x''_{k+1}	$<^1$	$<^2$
1	10.1843	12.9872	0	0.978448	0.489224	11.1718	12.1593	t	t
2	11.1718	9.00121	0	0.13702	0.06851	11.1551	11.1384	f	t
3	11.1384	8.97503	0	0.133119	0.0665595	11.1285	11.1186	f	t
4	11.1186	8.96543	0	0.1308	0.0654	11.1125	11.1064	f	t
5	11.1064	8.96174	0	0.129378	0.064689	11.1026	11.0988	f	t
6	11.0988	8.96028	0	0.12849	0.0642448	11.0964	11.094	f	t
7	11.094	8.9597	0	0.127929	0.0639643	11.0925	11.091	f	t
8	11.091	8.95946	0	0.127572	0.0637858	11.09	11.089	f	t
9	11.089	8.95936	0	0.127344	0.0636718	11.0884	11.0878	f	t
10	11.0878	8.95932	0	0.127198	0.0635988	11.0874	11.087	f	t
11	11.087	8.9593	0	0.127104	0.0635519	11.0867	11.0865	f	t
12	11.0865	8.9593	0	0.127044	0.0635218	11.0863	11.0861	f	t
13	11.0861	8.95929	0	0.127005	0.0635024	11.086	11.0859	f	t
14	11.0859	8.95929	0	0.12698	0.0634899	11.0859	11.0858	f	t
15	11.0858	8.95929	0	0.126964	0.0634819	11.0857	11.0857	f	t
16	11.0857	8.95929	0	0.126953	0.0634767	11.0857	11.0856	f	t
17	11.0856	8.95929	0	0.126947	0.0634734	11.0856	11.0856	f	t
18	11.0855	8.95929	0	0.126935	0.0634674	11.0855	11.0855	f	f
19	11.0855	8.95929	0	0.0634674	0.0317337	11.0855	11.0855	f	f
20	11.0855	8.95929	0	0.0317337	0.0158668	11.0855	11.0855	f	f
21	11.0855	8.95929	0	0.0158668	0.00793342	11.0855	11.0855	f	f
22	11.0855	8.95929	0	0.00793342	0.00396671	11.0855	11.0855	f	f

FIGURE 4.16: Box ratio steepest descent

$c(x)$ using interval arithmetics returns an an interval. The algorithm makes its checks with the lower and upper bounds of $c(x)$ and returns *true* if the constraint is violated. In the constraint type "c_i = 0", if $0 \notin c(x)$, then the constraint is violated. In "$c_i \leq 0$", the check is on the lower bound of $c(x)$ such that if $c(x).lb > 0$, then the constraint is violated.

`getPenalty` uses the function `isViolated(c,x)` to check if the constraint c_i is violated at the point x. If it is *true*, the penalty value is added to the summation variable $penalty$. The square value is mainly used to assure that no negative values are added to the overall penalty value. This can be caused by the constraints of type $c_i = 0$ if the value $c(x) < 0$.

In the second algorithm, `isViolated`, the check of a single constraint violation is direct and easily readable. If the constraint is an equality constraint, then an equality check is performed. If the constraint is an inequality constraint, then we check if this inequality is not satisfied.

After showing and discussing the box ratio with separate penalty steepest descent, we put it to the test by applying it to example 4.2 which is mentioned in section 4.1. The algorithm starts by applying a random point from the selected box b.

In table 4.16, the last two columns contain the check for `isBetterPoint`(x'_{k+1}, x'_{k+1}) and `isBetterPoint`(x_{k+1}, x_k), respectively. b has the interval $[10, 18.55]$ for the variable x. The selected random $x_k \in b$ is 10.1843 having $f(x_k) = 12.9872$. The nearest local

minimum to this position is the point $x = 11.0855$. We notice the ratio calculations with respect to the box. Since $\nabla f(x_k)$ is positive, then the ratio calculation is $\frac{18.55-10.18}{18.55-10} = 0.97$ giving a value of $x^{'}_{k+1} = 11.1718$ and $x^{''}_{k+1} = 12.1593$.

The algorithm chooses $x^{'}_{k+1}$ to be x_{k+1}. We notice how the value of the step length $ratio$ decreases as x_k converges into a local minimum. This process occurs until five successive *false* returns from the `isBetterPoint`(x_{k+1}, x_k) function is reached. The average time taken by the algorithm to find the optimal minimum of the problem is 0.04 seconds after pruning 4 boxes.

In some examples with a big number of constraints, as we will show in the next chapter, the newly obtained point x_{k+1} might go outside of the box b with the successive subtraction of the term **(1-r)** $\times\ ratios[j] \times \nabla c_j$ with every violated constraint $c \in C$.

In the modified version of the steepest descent with separate penalty and box ratio, a small part of code was added to ensure that the obtained x_{k+1} does not go outside of the box b for any variable interval in b. This version appears in figure 4.17.

The modification is added in line 23 of original algorithm, figure 4.17. Inside the for-loop, which loops over the variables in the problem, it checks for every variable in $x^{'}_{k+1.j}$ and $x^{''}_{k+1.j}$ if its value is in the bounds of b_j.

```
1  if x'_{k+1.j} < b_j.lb then
2  |   x'_{k+1.j} ← b_j.lb;
3  end
4  if x''_{k+1.j} < b_j.lb then
5  |   x''_{k+1.j} ← b_j.lb;
6  end
7  if x'_{k+1.j} > b_j.ub then
8  |   x'_{k+1.j} ← b_j.ub;
9  end
10 if x'_{k+1.j} > b_j.ub then
11 |   x'_{k+1.j} ← b_j.ub;
12 end
```

FIGURE 4.17: **Modified** Box Ratio with Separate Penalty Steepest Descent Algorithm

This modification helps in the comparison between using this algorithm with strictly bounding the selected points to be from inside the box b, or allowing the algorithm to choose a point outside of b.

In the example in figure 4.2, this modification will not change the output of the selected box. However, in the next chapter, the effect of this modification will appear, specially in the number of pruned boxes.

In conclusion, box ratio with dependant penalty algorithm provides a more modular technique of local search, meaning that the penalty calculation is separated from the objective function. The algorithm gives more priority to satisfy the constraints, as the

weight of moving into the negative direction of the gradient of the violated constraint is 0.75 of the step length, compared to 0.25 for the gradient of the objective function.

The step length calculation is depending on the width of the box and the sign of the objective function gradient. This gives a more greedy nature to the algorithm to converge faster into a local minimum in the box. We have the option to keep all the selected x_k-s to be in the input box b, or it can also be outside of b.

5

Testing and Analysis

In this chapter, we present the testing conducted on the algorithms mentioned in the previous chapter. The collected results are shown and analysed. We try to investigate which algorithm has a better overall timing. We detect the convergence rate to the global minimum and the number of needed branches using Procure's branching algorithm.

Throughout this chapter, two main examples will be used, the *Dipigri* problem [14], and the *HS108* problem [48]. Afterwards, we show our results on running a subset of the benchmarks that appears in [48]. This set of benchmarks was developed by Vanderbei as a part of testing the Optimization language AMPL. It is a standard set of benchmarks of non-linear optimization models including a collection of models that vary in the number of variables and constraints. Moreover, they vary in constraints types so they give the possibility of choosing constrained or unconstrained optimization problems. We discuss and analyse the gathered results, obtaining a clear conclusion for the comparisons between the different algorithms.

For each of the upcoming two sections, the problem is described and then the results of running the different algorithms over this problem are presented. We control the termination of the algorithms with the relativity conditions. We mention these relativity conditions in equation 5.1, since most of the problems we will use for testing are known landmark problems. We can use the available knowledge about the global minimum from these problems to construct the relativity condition, which stops the search when this condition is met:

$$\text{Algorithm stops when:} \quad \begin{cases} |f(x^{'}) - f(x^{*})| \leq |f(x^{*})| \times \mathbf{r}, & |f(x^{*})| > 1 \\ |f(x^{'}) - f(x^{*})| \leq \mathbf{r}, & |f(x^{*})| \leq 1. \end{cases} \tag{5.1}$$

```
var x{i in 1..7} := [-10..10];

minimize f:
  (x[1]-10)^2+5*(x[2]-12)^2+x[3]^4+3*(x[4]-11)^2+
  10*x[5]^6 + 7*x[6]^2+x[7]^4-4*x[6]*x[7] -10*x[6]-8*x[7];

subject to cons1:
  2*x[1]^2+3*x[2]^4+4*x[4]^2+x[3]+5*x[5]-127.0 <= 0;
subject to cons2:
  10*x[3]^2+7*x[1]+3*x[2]+x[4]-x[5]-282.0 <= 0;
subject to cons3:
  x[2]^2+6*x[6]^2+23*x[1]-8*x[7]-196.0 <= 0;
subject to cons4:
  4*x[1]^2+x[2]^2-3*x[1]*x[2]+2*x[3]^2+5*x[6]-11*x[7] <= 0;
```

FIGURE 5.1: Example 2, Dipigri optimization problem

$f(x^*)$ is the known global minimum value. $f(x^{'})$ is the minimal value found by the local search algorithm so far. $\mathbf{r} \in [0, 1]$ is a multiplication coefficient which controls how close is $f(x^{'})$ from $f(x^*)$, that is the acceptable range to stop the algorithm.

The first equation is the concept of relative error which is a widely used stopping criterion. It calculates an approximate of a value and normalises it over the original value. The second equation is the absolute error which calculates the distance between two values.

The **r** value is inversely proportional with the accuracy of the obtained $f(x^{'})$. When **r** increases, the acceptable range becomes wider and the accuracy decreases. In our tests, we will use different values for **r**, to compare the convergence rates of different algorithms over several acceptable ranges.

5.1 First Optimization Problem: Dipigri

The Dipigri problem is presented in [14]. It is a minimization problem with *seven* variables, which are subject to *four* constraints. Figure 5.1 shows the problem in AMPL syntax. AMPL [15] is an abbreviation for A Mathematical Programming Language syntax. Procure includes an AMPL parser that parses the AMPL problem into Procure structure. The syntax of AMPL is direct and easily understandable.

The global minimum of the Dipigri problem is 680.6301. In table 5.2, we show the result of the average of *ten* runs of the algorithms appeared in the previous chapter on the given problem. We do that by displaying the average time in *seconds* and the average best local minimum reached by every algorithm.

Four values of **r** are selected. *MIDP* is the Procure's default mid-point algorithm mentioned in 4.1. *RAND* is the random local search algorithm. *AQSD* is the Armijo rule with Quadratic penalty Steepest Descent. *Mod. AQSD* is the modified version of it, shown on figure 4.9. *BRSP* is the Box Ratio with Separate Penalty steepest descent algorithm. *Mod. BRSP* is the modification of it, as it can be seen on figure 4.17.

	$\mathbf{r} = 0.2$		$\mathbf{r} = 0.1$		$\mathbf{r} = 0.01$		$\mathbf{r} = 0.005$	
Algorithm	*time (s)*	$f(x^{'})$	*time (s)*	$f(x^{'})$	*time (s)*	$f(x^{'})$	*time (s)*	$f(x^{'})$
MIDP	1.13	824.3	3.25	755.9	171.4	687.4	354.6	683.1
RAND	1.28	800.2	1.89	728.7	28.6	686.2	80.92	683.7
AQSD	2.42	780.7	3.14	740.4	437.6	685.5	1515	682.1
Mod. AQSD	2.39	805.7	4.84	738.4	207.1	686.7	1163	683.1
BRSP	1.15	754.4	1.80	710.8	2.96	685.8	10.13	682.8
Mod. BRSP	1.02	721.5	1.91	733.4	3.06	686.5	8.87	683

FIGURE 5.2: Results of the five different algorithms on Dipigri Problem

For $\mathbf{r} = 0.2$, meaning that the acceptable range is in 20% around the global minimum $f(x^*)$, we find that most of the algorithms find the solution in the first 3 seconds, with a very slight latency in the *AQSD* algorithm and its modification. All the algorithms converge very fast in the beginning around the global minimum area. This shows the strength of Procure's propagation algorithms to prune the boxes. Hence, the prune was successful to eliminate around 80% of the search space in the first second.

The difference starts when $\mathbf{r} = 0.1$. For the random algorithm, it took an average of 1.89 seconds to find a close value to the global minimum. This is better than the *MIDP* average time. This shows that trying many random points in a box gives a higher potential to find a feasible solution faster than getting the mid-point of the box. Moreover, we notice that the *AQSD* and its modification are taking slightly more time than the *RAND* and the *BRSP* algorithms. For the *BRSP* algorithm with $\mathbf{r} = 0.1$, we see that it also gets the solution almost as fast as $\mathbf{r} = 0.2$. The same applies for the *Mod. BRSP* algorithm. Both of the algorithms are having almost the same average time of running. This shows the convergence speed of the algorithm when we separate the penalty calculation from the objective function.

For $\mathbf{r} = 0.01$, we see a very noticeable time difference between the algorithms. It took an average of 28.6 seconds for the random algorithm to find a solution. However, for the *AQSD* algorithm, the time taken is significantly more, with 437.6 seconds for the original method. This is even slower than the normal mid-point algorithm provided by Procure. This phenomena can be explained by the time taken to calculate the new objective function Q, for calculating the penalty function. Also, the time taken for converging to better solutions, since the stopping criteria does not guarantee a stop if the newly obtained x_{k+1} is very close, almost identical to x_k.

Moreover, for the *Mod. AQSD*, the resulting time is noticeably better than the original method. Yet, it is still slower than the mid-point and random algorithms. This is mainly due to the amount of calculations performed, which increases the complexity of the program, compared to the simplicity of the mid-point algorithm and the random search.

BRSP algorithm with $\mathbf{r} = 0.01$ performs visibly better than the other algorithms. This is due to the separation of penalty calculation and function calculations. Moreover, the greedy step length taken as the distance between the box bound and the current position

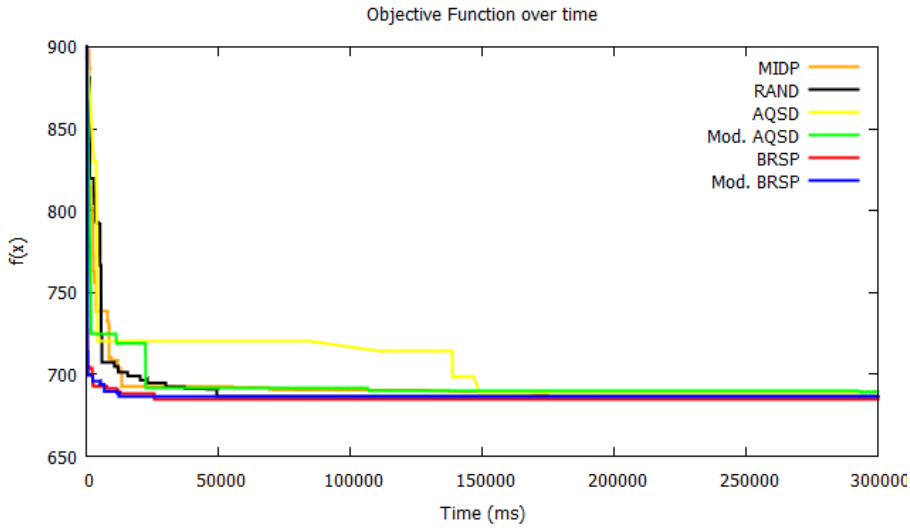

FIGURE 5.3: Dipigri problem algorithms convergence over time

of x_k in the box plays a major role. This creates a large step taken by the algorithm in one iteration, compared to the *AQSD* algorithm. *Mod. BRSP* is showing similar time as the *BRSP*. This can be explained as follows: the propagation strength of branching and pruning tightens the search space to include the feasible area and exclude the infeasible area as much as possible, which decreases the possibility of finding a valid solution when x_{k+1} goes outside of the box.

RAND takes an average of 80.92 seconds to reach a solution with $\mathbf{r} = 0.005$, which is a good time compared to the *AQSD* algorithm. Considering the accuracy required with the value of $\mathbf{r}$, *AQSD* spends more time doing more calculations and slowly converges to the global minimum.

With larger greedy steps and separation of the penalty calculation and the objective functions, we find *BRSP* algorithm even faster than the random search. Moreover, by not allowing the obtained x_{k+1} in *Mod. BRSP* to be outside of the box, performance gets better with 8.87 seconds on average to obtain a global minimum for this problem.

In conclusion, on the runtime of the problem on the different algorithms we notice that the algorithms yield a good time when the complexity is lower. As our current example shows, the random algorithm achieved better time than *AQSD* algorithm due to the complexity of the calculations involved in the *AQSD* algorithms.

BRSP is generally having better performance for this problem than the other algorithms. This can be explained by the greediness of the step length and the less amout of calculations needed. *Mod. BSRP* is having the best time on very small values of $\mathbf{r}$. Therefore we can conclude that separating the penalty calculations from the objective function is a better technique than getting a quadratic penalty function Q for this example.

In figure 5.3, we show the convergence of the algorithms over time to reach the obtained minimum with $\mathbf{r} = 0.01$. We select a run from the 10 runs we did for every algorithm. Notice that all the algorithms except the *AQSD* algorithm converge to a very

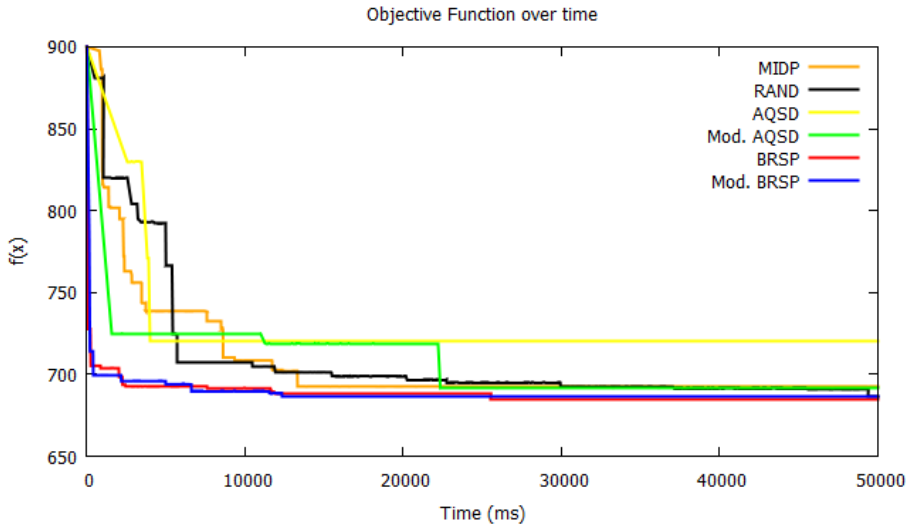

FIGURE 5.4: Dipigri problem algorithms convergence over the first 50 seconds

acceptable area in the first 50 seconds. The *AQSD* takes more time due to the huge number of iterations when the new x_{k+1} is very close x_k.

We zoom in on the first 50 seconds in figure 5.4. We observe the fastest algorithm to converge are the *BRSP* and *Mod. BRSB*. However, the *Mod. AQSD* converges to be very close to the acceptable area ($\mathbf{r} = 0.0061$), nearly the same area of *RAND* and *BRSP*.

Next we discuss every algorithm with $\mathbf{r} = 0.01$. In figure 5.5, we show the average number of pruned boxes for the Dipigri problem and we calculate the average time taken per box.

The *MIDP* algorithm needs 53659 box splits in order to find a solution in the acceptable range. This is due to the lack of performing any kind of search for the problem. Therefore, it mainly depends on the power of branching and pruning of the Procure framework. However, due to the simplicity of the calculations needed to calculate a box's mid-point, the average time spent per box is around 3 milliseconds. This gives the algorithm its ability to compensate the huge number of splits with a very small time spent in one box.

For the *RAND* algorithm, the random selection of the algorithm for each box is very low in complexity. The only check done is to decide if the obtained random point x is feasible or not. This explains the very short time needed per every box which is around 5 milliseconds. The average number of boxes needed to be split and checked are 5934.4 boxes. This shows that it needed a less amount of splits and propagations to find an acceptable solution than the *MIDP* algorithm. This is because inside each box, we try several points as a potential solution instead of only the mid-point.

For the *AQSD*, we notice the latency during the checking inside a single box. It took an average of 2084 boxes to find a solution. However, the algorithm on average took 210 milliseconds per box in order to converge to the box's minimum. Moreover, it might not reach a solution in the box, since it can keep converging into a point having a better $Q(x)$,

algorithm	*Boxes*	*Av. t per box (msec.)*
MIDP	53659	3
RAND	5934	5
AQSD	2084	210
Mod.AQSD	1874	110
BRSP	182	16
Mod.BRSP	157	19

FIGURE 5.5: Dipigri average number of branched boxes and the average time per box $\mathbf{r} = 0.01$

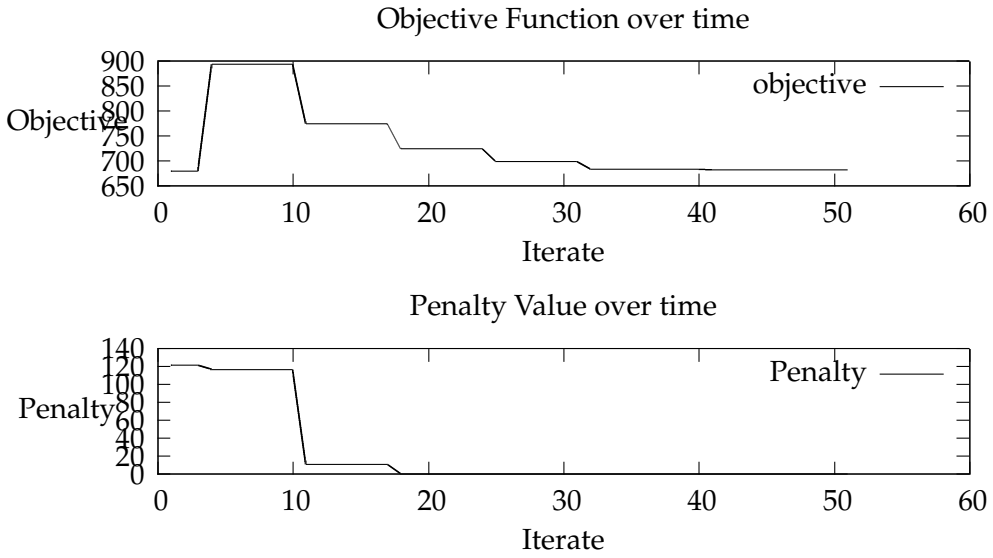

FIGURE 5.6: Objective function vs Penalty over time in *BRSP*

.

but one which is not a feasible point. We will discuss this more in figure 5.7.

Mod.AQSD eliminates that huge amount of iterations due to the modified stopping criteria, reducing the overall time by about 40%. However, we notice in the *Mod. AQSD*, that after 33 seconds, the algorithm converged to a value of $r = 0.012$.

The *Mod. BRSP* has the best solution timings with an average of 156 boxes branched, with an average of 19 milliseconds per box. This happens due to the greedy technique of the step length selection. It also restricts the selected points to be inside the box. This shows the power of the *BRSP* and its modification to find a feasible solution inside a box in a very fast time. Thus making it perform lower number of splits, which improves the overall time taken by the algorithm.

We see in figure 5.6 the iteration inside a box using the *BRSP* problem. The box b presented is a box where the final solution for the Dipigri problem was found. When the iterations start on b, the point x_k has a very high penalty value. In the next iterations, the algorithm is more concerned with decreasing the penalty value more than the objective function value. Which in turn causes a decrease in the penalty value, not taking into account the value of the objective function.

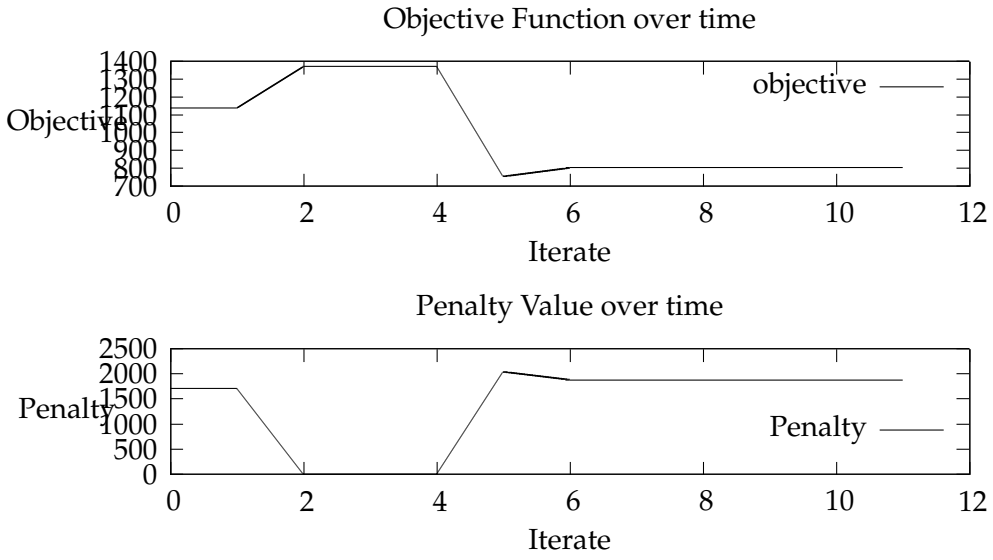

FIGURE 5.7: Objective function vs Penalty over time in *AQSD*

.

In later iterations, we notice the penalty value becoming 0 around the 18^{th} iteration. After this point, the algorithm is concerned only in obtaining new points with a lower objective function value. This happens until eventually a point with an objective function value close to the global minimum is obtained.

On the other hand, in figure 5.7 we show the iterations of the *Mod.AQSD* over one of the boxes with $\mathbf{r} = 0.01$. However, instead of showing the value of $Q(x_k)$, we show the two values of the normal objective function $f(x_k)$ and the penalty value of x_k, which is calculated using the function `getPenalty(`x`)` shown in figure 4.15.

We notice that in the first and second iterations, the penalty value decreases to zero. However, in the next two iterations, a new point x_{k+1} is tried having $f(x_{k+1}) = 752.7$ and $penalty(x_{k+1}) = 2035$. With the increasing value of μ over every iteration, the penalty value is decreased to a quarter of its value.

This happens until in the 5^{th} iteration, when $Q(x_{k+1}) = 752 + \frac{1}{2\mu} \times 2035$ and $\mu = 2$. Causing the value of $Q(x)$ to be 1007.07 which is less than $Q(x_k) = 1373$. Therefore, x_{k+1} is the new x_k. However, the algorithm does find a new point which is a feasible point. In conclusion, the box fails to return a feasible point. This increased latency of finding a feasible point, in turn increases the number of checked boxes.

5.2 Second Optimization Problem : HS108

The HS108 is a problem from the benchmark set of Vanderbei, it appears in [48]. This problem has *nine* variables and *fourteen* constraints. This is a bigger number than the ones of the Dipigri problem. The problem is having a global minimum of -0.866. Therefore, the absolute error technique is used from equation 5.1. We show the problem in *AMPL*

```
var x {1..9};

minimize obj:
   -.5*(x[1]*x[4]-x[2]*x[3]+x[3]*x[9]-x[5]*x[9]+x[5]*x[8]-x[6]*x[7]);

s.t. c1:  1-x[3]^2-x[4]^2>=0;
s.t. c2:  1-x[5]^2-x[6]^2>=0;
s.t. c3:  1-x[9]^2>=0;
s.t. c4:  1-x[1]^2-(x[2]-x[9])^2>=0;
s.t. c5:  1-(x[1]-x[5])^2-(x[2]-x[6])^2>=0;
s.t. c6:  1-(x[1]-x[7])^2-(x[2]-x[8])^2>=0;
s.t. c7:  1-(x[3]-x[7])^2-(x[4]-x[8])^2>=0;
s.t. c8:  1-(x[3]-x[5])^2-(x[4]-x[6])^2>=0;
s.t. c9:  1-x[7]^2-(x[8]-x[9])^2>=0;
s.t. c10: x[1]*x[4]-x[2]*x[3]>=0;
s.t. c11: x[3]*x[9]>=0;
s.t. c12: -x[5]*x[9]>=0;
s.t. c13: x[5]*x[8]-x[6]*x[7]>=0;
s.t. c14: x[9]>=0;
```

FIGURE 5.8: Example 3, HS108 optimization problem

	$\mathbf{r} = 0.3$		$\mathbf{r} = 0.2$		$\mathbf{r} = 0.1$	
Algorithm	*time*	$f(x^{'})$	*time*	$f(x^{'})$	*time*	$f(x^{'})$
MIDP	0.90	-0.722	0.88	-0.723	21.71	-0.780
RAND	0.64	-0.618	1.78	-0.686	11.03	-0.7825
AQSD	18.70	-0.694	44.34	-0.688	>1000	> -0.6
Mod. AQSD	5.58	-0.597	27.52	-0.694	>1000	> -0.6
BRSP	0.11	-0.594	0.67	-0.697	6.08	-0.782
Mod. BRSP	0.12	-0.629	0.65	-0.686	3.99	-0.77

FIGURE 5.9: Average results of the five different algorithms on HS108 problem

syntax in figure 5.8. We select three values of $\mathbf{r}$. $\mathbf{r} = 0.3$, $\mathbf{r} = 0.2$ and $\mathbf{r} = 0.1$.

We run the problem on the six algorithms presented in the previous chapter. We run every algorithm ten times. The results of the average time and obtained objective functions appear in figure 5.9.

For $\mathbf{r} = 0.3$, meaning that $|f(x^{'}) - f(x^{*})| \leq 0.3$, we notice that the *AQSD* algorithm takes a very noticeably longer average time than the other algorithms. This is explained due to the stopping criteria that does not stop if the newly obtained point x_{k+1} in the box is very close to the point x_k. It gets better with only 5.58 seconds with the *Mod.AQSD* algorithm. However, the other algorithms score a better average time under 1 second.

For $\mathbf{r} = 0.2$, we notice a very slight difference between the average of the *MIDP* and *RAND* algorithms. However, the more noticeable difference is the one of the *AQSD*. This shows that the convergence to a minimal point is very slow, taking a lot of iterations which increase the time to solve the problem. The *Mod.AQSD* decreases the time by about 40%. This is very slow compared with the average timings obtained by the other algorithms.

The *BSRP* algorithm and its modification shows better average time than all the other

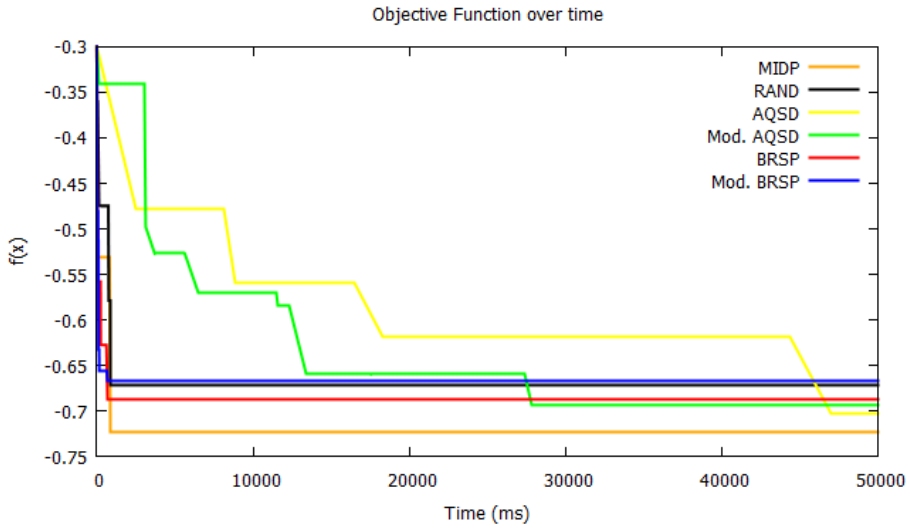

FIGURE 5.10: HS108 problem algorithms convergence over time with $\mathbf{r} = 0.2$

algorithms. This shows that the separation of penalty from the calculation of the objective function and having greedy step lengths to the edge of the box converges faster into a minimal point.

The average time of *AQSD* and its modification are over 1000 seconds when $\mathbf{r} = 0.1$. Which means that it is very hard to find a feasible point when we use the *AQSD*. We will clarify this in figure 5.12. The *BSRP* algorithm took 6.08 seconds to find a solution with $\mathbf{r} = 0.1$. With the modification *Mod. BRSP*, this time is almost halved which shows the power of Procure's constraint propagation techniques, and that the feasible solutions are inside a pruned box.

In figure 5.10, we show the convergence of the best minimal point $f(x^{'})$ over time until it reaches an absolute error less than or equal to 0.2. We observe the very slow convergence rate of both the *AQSD* and *Mod.AQSD*.

We notice also the very quick convergence of all the other algorithms to the acceptable range in the first second. In figure 5.11, we show the first second of figure 5.10. *BRSP* and its modification converges first to the acceptable area, then followed by the *RAND* and *MIDP* algorithms.

In this problem, we see again the advantage gained by separating the penalty from the objective function and using the long step lengths to the edge of the box in *BRSP* and its modification. It allows fast convergence to the box's local minimum. Moreover, the focus on reducing the penalty value of the point x_k first, then reducing the objective function makes the algorithm find a feasible point more frequently than the *AQSD* algorithms.

Thus, with the *Mod. BRSP* and relatively small values of $\mathbf{r}$, we get the best convergence rate to the acceptable areas. This shows again the strength of the Procure constraint propagation techniques to prune the boxes so that global minimum exists in one of the pruned boxes.

On the other hand, we notice that the quadratic penalty technique is not as efficient

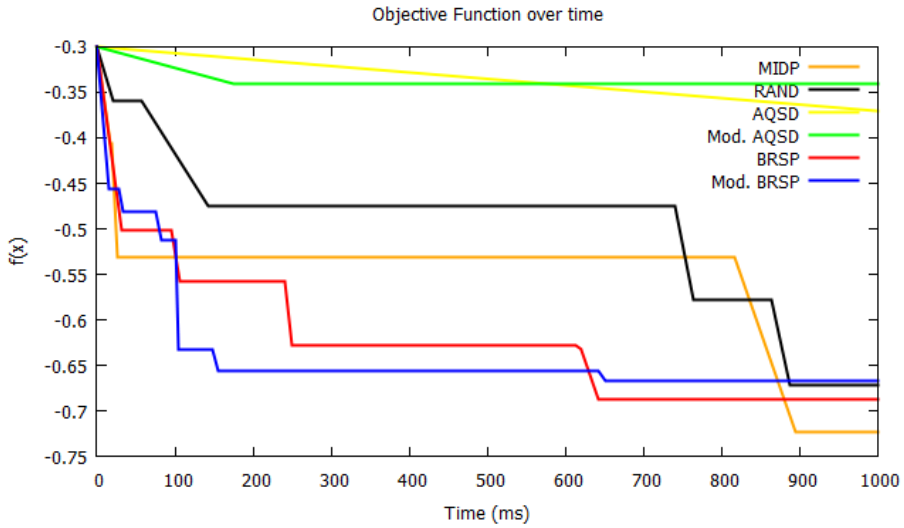

FIGURE 5.11: Dipigri problem algorithms convergence over the first second

as separating the penalty from the objective function. As this can cause the search to fail to find a feasible point in several boxes, which delays the constraint propagations done by Procure. Having more control over the stopping criteria in *Mod. AQSD* gives better timing, as it cuts off a big number of iterations. However, this does not solve the main problem of the quadratic penalty approach, because the difficulty of finding a feasible solution stays the same.

In figure 5.12. we show the iterations of the *AQSD* over one of the boxes with $\mathbf{r} = 0.2$. We show the two values of the normal objective function $f(x_k)$ and the penalty value of x_k, the same as figure 5.7.

We notice again the same as the Dipigri problem how the algorithm fails to find a feasible solution in this box even if it found a feasible point twice, at the 2^{nd} and 5^{th} iterations. This shows that with the *AQSD* algorithm and its modification, it is not guaranteed that if a feasible point is found in a box, it will be returned as the solution.

Therefore, in the *HS108* problem, the technique of the *BRSP* algorithm scores the best times and the modified version *Mod.BRSP* has the fastest time. Moreover, The *AQSD* algorithm is the slowest technique to solve this problem.

5.3 Benchmarks and Comments

In this section, we show the result of running a set of the benchmarks that appears in [48]. We changed the stopping criteria of the program: instead of giving the error value $\mathbf{r}$ and wait for the algorithm to reach this $\mathbf{r}$. We have two controllers as the search stops when it exceeds the time t or when a specific error value $\mathbf{r}^{'}$ reached.

We run the benchmarks with $t = 60$ seconds and $\mathbf{r}^{'} = 0.0001$. We run every problem for the six algorithms ten times each. We calculate the average for each algorithm and show the results in the following table.

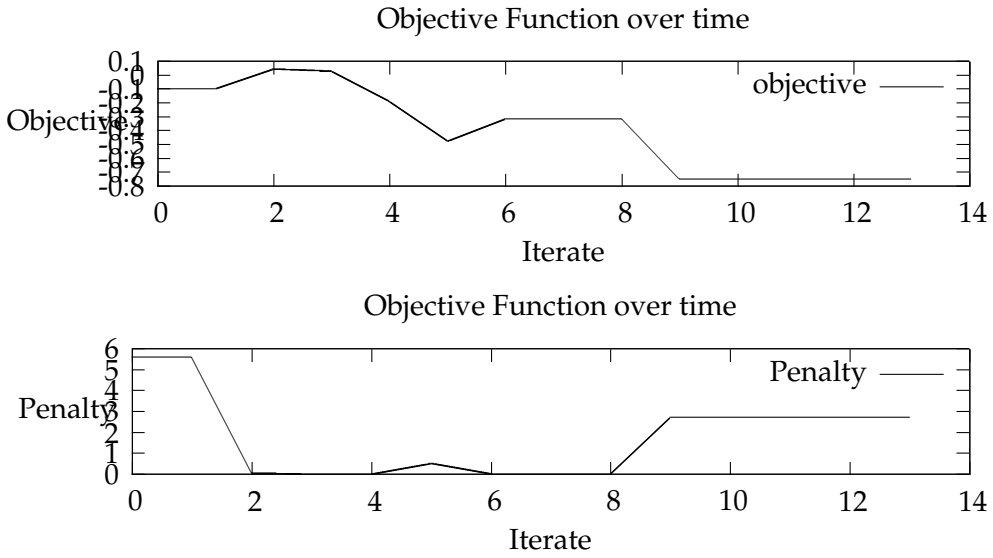

FIGURE 5.12: HS108 objective function vs Penalty over time in *AQSD*

.

Problem	*V*	*C*	$f(x^*)$	$f(x^{'})$	*Boxes*	**r**	*time*
allinitMIDP	3	0	5.7444	16.7073	5793	0.0001	12.184
allinitRAND	3	0	5.7444	16.707	4057.7	0.0001	10.1098
allinitAQSD	3	0	5.7444	16.706	1.8333	0.0001	1.4055
allinitModAQSD	3	0	5.7444	16.7071	43.8	0.0001	2.3422
allinitBRSP	3	0	5.7444	16.706	119.4	0.0001	1.1929
allinitModBRSP	3	0	5.7444	16.706	3	0.0001	0.135
allinituMIDP	4	0	5.7444	5.7891	4459	0.0077	60
allinituRAND	4	0	5.7444	5.7839	4339.8	0.0068	60
allinituAQSD	4	0	5.7444	168	22.5	32.6	60
allinituModAQSD	4	0	5.7444	5.7509	396.7	0.0011	60
allinituBRSP	4	0	5.7444	5.7444	556.9	0.0001	10.552
allinituModBRSP	4	0	5.7444	5.7444	55.8	0.0001	2.0245
biggs3MIDP	3	0	0	3.1045	31815	3.1045	60
biggs3RAND	3	0	0	2.2463	20473.8	2.2463	60
biggs3AQSD	3	0	0	0	30.8	0.0001	38.4692
biggs3ModAQSD	3	0	0	0.2017	693.7	0.2018	44.101
biggs3BRSP	3	0	0	0.1424	1490.7	0.1424	24.2823
biggs3ModBRSP	3	0	0	0.4259	4052.5	0.4259	60
biggs5MIDP	5	0	0.0056	9.8637	4274	9.8581	60
biggs5RAND	5	0	0.0056	0.5998	1730.7	0.5942	60
biggs5AQSD	5	0	0.0056	8.93	100.8	8.9244	60
biggs5ModAQSD	5	0	0.0056	6.9961	192.5	6.9905	60
biggs5BRSP	5	0	0.0056	0.4491	1117.1	0.4435	60
biggs5ModBRSP	5	0	0.0056	0.2725	457.9	0.2669	60
biggs6MIDP	6	0	0	9.8637	5542	9.8637	60
biggs6RAND	6	0	0	8.7449	2974.1	8.7449	60

Table 5.1 .. *Continued from previous page*

Problem	*V*	*C*	$f(x^*)$	$f(x^{'})$	*Boxes*	**r**	*time*
biggs6AQSD	6	0	0	9.7479	147.5	9.7479	60
biggs6ModAQSD	6	0	0	9.8562	156.5	9.8562	60
biggs6BRSP	6	0	0	7.9975	1669.2	7.9975	60
biggs6ModBRSP	6	0	0	0.3064	474.1	0.3064	60
cliffMIDP	2	0	0.1998	0.1998	1953	0.0001	19.748
cliffRAND	2	0	0.1998	0.1999	1704.5	0.0001	16.6891
cliffAQSD	2	0	0.1998	0.1998	120	0.0001	16.4268
cliffModAQSD	2	0	0.1998	0.1998	208	0.0001	5.8508
cliffBRSP	2	0	0.1998	0.1998	131.4	0.0001	4.7205
cliffModBRSP	2	0	0.1998	0.1998	139	0.0001	4.7638
denschndMIDP	3	0	0	0	1071	0.0001	41.683
denschndRAND	3	0	0	0.0001	410.7	0.0001	11.0175
denschndAQSD	3	0	0	678	43.7	678	60
denschndModAQSD	3	0	0	16x5	325.8	16x6	60
denschndBRSP	3	0	0	0	316.1	0.0001	10.9872
denschndModBRSP	3	0	0	0	457.1667	0.0001	17.5355
dipigriMIDP	7	4	680.6301	692.798	16586	0.0176	60
dipigriRAND	7	4	680.6301	684.6948	11860.5	0.0059	60
dipigriAQSD	7	4	680.6301	694.2605	510.4	0.0196	60
dipigriModAQSD	7	4	680.6301	695.4887	527.5	0.0213	60
dipigriBRSP	7	4	680.6301	682.4481	3401.9	0.0027	60
dipigriModBRSP	7	4	680.6301	682.0021	2166.3	0.0012	60
ex3.1.4MIDP	3	3	-4	-3.9996	8625	0.0001	16.24
ex3.1.4RAND	3	3	-4	-3.9997	8136.5	0.0001	15.8626
ex3.1.4AQSD	3	3	-4	-3.9424	16.4	0.0147	50.7582
ex3.1.4ModAQSD	3	3	-4	-3.6544	39.1	0.1243	60
ex3.1.4BRSP	3	3	-4	-4	34.3	0.0001	1.8473
ex3.1.4ModBRSP	3	3	-4	-3.8471	577.2	0.0405	60
ex8.5.3MIDP	5	5	-0.0042	0.0006	7042	0.0048	60
ex8.5.3RAND	5	5	-0.0042	-0.0024	6647.3	0.002	60
ex8.5.3AQSD	5	5	-0.0042	1×10^{7}	13	1×10^{7}	60
ex8.5.3ModAQSD	5	5	-0.0042	1×10^{7}	13	1×10^{7}	60
ex8.5.3BRSP	5	5	-0.0042	0.0777	138.6	0.0819	60
ex8.5.3ModBRSP	5	5	-0.0042	0.0778	852.8	0.082	60
ex8.5.4MIDP	5	4	-0.0004	-0.0001	4293	0.0003	60
ex8.5.4RAND	5	4	-0.0004	0.1367	3716.5	0.1371	60
ex8.5.4AQSD	5	4	-0.0004	8×10^{5}	13.1	8×10^{5}	60
ex8.5.4ModAQSD	5	4	-0.0004	1×10^{7}	13.8	1×10^{7}	60
ex8.5.4BRSP	5	4	-0.0004	0.5694	193.6	0.5698	60
ex8.5.4ModBRSP	5	4	-0.0004	0.4765	331.1	0.4769	60
expfitbMIDP	5	101	0.005	21387.9	1145	21387.9	60
expfitbRAND	5	101	0.005	3232.8793	562.5	3232.8743	60
expfitbAQSD	5	101	0.005	749999.25	0.75	749999.25	60
expfitbModAQSD	5	101	0.005	1×10^{7}	1	1×10^{7}	60
expfitbBRSP	5	101	0.005	2024	21.6	2024	60

Table 5.1 .. *Continued from previous page*

Problem	*V*	*C*	$f(x^*)$	$f(x')$	*Boxes*	**r**	*time*
expfitbModBRSP	5	101	0.005	1940	15.7	1940	60
genhumpsMIDP	5	0	0	0.0001	3637	0.0001	31.569
genhumpsRAND	5	0	0	0.0001	1072.8	0.0001	21.4408
genhumpsAQSD	5	0	0	0	1.5	0.0001	2.914
genhumpsModAQSD	5	0	0	0	4.7	0.0001	3.3894
genhumpsBRSP	5	0	0	0	8.1	0.0001	0.8991
genhumpsModBRSP	5	0	0	0	8.4	0.0001	0.9687
haifasMIDP	7	9	-0.45	30.0492	5941	30.4992	60
haifasRAND	7	9	-0.45	9×10^5	5985.6	9×10^5	60
haifasAQSD	7	9	-0.45	0.8859	8.6	1.3359	60
haifasModAQSD	7	9	-0.45	-0.0134	12.1	0.4366	60
haifasBRSP	7	9	-0.45	-0.4451	1430.2	0.0049	60
haifasModBRSP	7	9	-0.45	-0.3572	104.2	0.0928	60
himmelbfMIDP	4	0	318.5717	5434.38	1150	0.9414	60
himmelbfRAND	4	0	318.5717	2020.4956	1045.3	0.7955	60
himmelbfAQSD	4	0	318.5717	11142.038	14	0.9611	60
himmelbfModAQSD	4	0	318.5717	4213.4627	284.4	0.8948	60
himmelbfBRSP	4	0	318.5717	2649.6384	588	0.8355	60
himmelbfModBRSP	4	0	318.5717	1595.4241	366.4	0.7047	60
hs043MIDP	4	3	-44	-43.9536	10341	0.0011	60
hs043RAND	4	3	-44	-43.9819	9377.6	0.0004	60
hs043AQSD	4	3	-44	-43.4956	268.1	0.0116	60
hs043ModAQSD	4	3	-44	-43.771	594.1	0.0052	60
hs043BRSP	4	3	-44	-43.986	3720.6	0.0003	60
hs043ModBRSP	4	3	-44	-43.9314	2055	0.0016	60
hs086MIDP	5	6	-32.3487	-31.8958	14664	0.0142	60
hs086RAND	5	6	-32.3487	-31.9032	14920.1	0.014	60
hs086AQSD	5	6	-32.3487	-2	1.1	15.1	60
hs086ModAQSD	5	6	-32.3487	-25.0358	20.6	0.3245	60
hs086BRSP	5	6	-32.3487	-31.9804	3187.5	0.0115	60
hs086ModBRSP	5	6	-32.3487	-31.7267	3075.7	0.0196	60
hs100MIDP	7	4	678.7547	692.798	17064	0.0176	60
hs100RAND	7	4	678.7547	684.8632	13053.2	0.0062	60
hs100AQSD	7	4	678.7547	693.7088	543.3	0.0188	60
hs100ModAQSD	7	4	678.7547	694.5121	564.1	0.02	60
hs100BRSP	7	4	678.7547	682.713	3405.1	0.0031	60
hs100ModBRSP	7	4	678.7547	682.5004	1606.8	0.0027	60
hs100modMIDP	7	4	678.7547	685.772	30380	0.0102	60
hs100modRAND	7	4	678.7547	680.6367	25094.3	0.0028	60
hs100modAQSD	7	4	678.7547	687.2776	1092.6	0.0124	60
hs100modModAQSD	7	4	678.7547	686.9273	1105	0.0119	60
hs100modBRSP	7	4	678.7547	680.1308	4465	0.002	60
hs100modModBRSP	7	4	678.7547	680.5883	2251.2	0.0027	60
hs108MIDP	9	13	-0.866	-0.7806	29575	0.0854	60
hs108RAND	9	13	-0.866	-0.8096	26001.7	0.0564	60

Table 5.1 .. *Continued from previous page*

Problem	*V*	*C*	$f(x^*)$	$f(x^{'})$	*Boxes*	**r**	*time*
hs108AQSD	9	13	-0.866	-0.7169	39.9	0.1491	60
hs108ModAQSD	9	13	-0.866	-0.706	142	0.16	60
hs108BRSP	9	13	-0.866	-0.8202	4248.6	0.0858	60
hs108ModBRSP	9	13	-0.866	-0.8127	3481.5	0.0533	60
hs113MIDP	10	8	24.3062	42.0184	31130	0.4215	60
hs113RAND	10	8	24.3062	37.5323	28219.4	0.3498	60
hs113AQSD	10	8	24.3062	391.9319	95.8	0.6954	60
hs113ModAQSD	10	8	24.3062	52.1355	151.6	0.5115	60
hs113BRSP	10	8	24.3062	27.5105	2160.6	0.1159	60
hs113ModBRSP	10	8	24.3062	28.7812	459.7	0.1539	60
leastMIDP	3	0	14085.1	25127.5	1878	0.4395	60
leastRAND	3	0	14085.1	25033.39	1946.1	0.4373	60
leastAQSD	3	0	14085.1	17188008.7	26.8	0.9852	60
leastModAQSD	3	0	14085.1	24783.01	426.5	0.4316	60
leastBRSP	3	0	14085.1	25602.99	1497.3	0.4492	60
leastModBRSP	3	0	14085.1	25175.4	1437.2	0.4404	60
minmaxrbMIDP	3	4	0	0	1445	0.0001	0.336
minmaxrbRAND	3	4	0	0.0001	1172.75	0.0001	0.2575
minmaxrbAQSD	3	4	0	349.6232	14.5	349.6232	60
minmaxrbModAQSD	3	4	0	311.302	21.9	311.302	60
minmaxrbBRSP	3	4	0	0.0001	51.8	0.0001	2.0725
minmaxrbModBRSP	3	4	0	0.0001	51.8	0.0001	1.9126
mistakeMIDP	9	13	-1	-0.9579	35713	0.0421	60
mistakeRAND	9	13	-1	-0.9638	38177.7	0.0362	60
mistakeAQSD	9	13	-1	-0.8002	27.9	0.1998	60
mistakeModAQSD	9	13	-1	-0.8113	137	0.1887	60
mistakeBRSP	9	13	-1	-0.9744	4613.8	0.0256	60
mistakeModBRSP	9	13	-1	-0.9799	3111.5	0.011	60
osborneaMIDP	5	0	0.0001	5579.52	0	5579.52	60
osborneaRAND	5	0	0.0001	12167791.5	739	12167791.5	60
osborneaAQSD	5	0	0.0001	1.106	1.9	1.1059	60
osborneaModAQSD	5	0	0.0001	1.1945	6.9	1.1944	60
osborneaBRSP	5	0	0.0001	171.9015	166.9	171.9014	60
osborneaModBRSP	5	0	0.0001	13.5723	0	13.5723	60
pspdocMIDP	4	0	2.4142	2.4144	1399	0.0001	8.795
pspdocRAND	4	0	2.4142	2.4144	561.9	0.0001	3.9262
pspdocAQSD	4	0	2.4142	12374.0959	83.6	0.7998	58.7231
pspdocModAQSD	4	0	2.4142	2.4143	231.3	0.0001	7.5099
pspdocBRSP	4	0	2.4142	2.4142	195.1	0.0001	11.8092
pspdocModBRSP	4	0	2.4142	2.4142	207.3	0.0001	13.3074
rosenmmxMIDP	5	4	-44	-18.3206	2889	1.4017	60
rosenmmxRAND	5	4	-44	106837.6811	3226	2.1608	60
rosenmmxAQSD	5	4	-44	2.6461	6	1.834	60
rosenmmxModAQSD	5	4	-44	-29.8421	18.4	0.5764	60
rosenmmxBRSP	5	4	-44	-42.7868	206.3	0.0287	60

Table 5.1 .. *Continued from previous page*

Problem	*V*	*C*	$f(x^*)$	$f(x^{'})$	*Boxes*	**r**	*time*
rosenmmxModBRSP	5	4	-44	-42.7881	192.9	0.0283	60
snakeMIDP	2	2	-0.0085	6251.7	9608	6251	60
snakeRAND	2	2	-0.0085	26014.404	8882.6	26014	60
snakeAQSD	2	2	-0.0085	3126.426	5.1	3126	60
snakeModAQSD	2	2	-0.0085	4668.011	83.1	4668	60
snakeBRSP	2	2	-0.0085	1982.9444	403.3	1982	60
snakeModBRSP	2	2	-0.0085	1290.3874	436.8	1290	60

TABLE 5.1: Results obtained from the benchmarks set

We tested our algorithms over 28 optimization problems. *V* is the number of variables in the problem. *C* is the number of constraints. We notice that the general pattern is that the *RAND* and *MIDP* algorithms are relatively close to each other with respect to $f(x^{'})$ and **r**. However, with respect to the average number of checked boxes, the *MIDP* algorithm have a very high number compared to the *algorithm*. This is due to the nature of the *RAND* that checks more points than the *MIDP* inside a box b which gives more potential for *RAND* to find a feasible point.

The benchmarks also confirm the results we obtained in the previous sections regarding the *AQSD* and the *Mod.AQSD* algorithms. Combining the constraints penalty and the objective function value into one function using the quadratic penalty technique is not efficient as it takes a long time for the calculation. We notice that *AQSD* and *Mod.AQSD* have the highest values of **r**. The average number of boxes checked by *AQSD* and its modification is very low compared to other algorithms. This is due to the long time taken for every box to converge slowly to a local minimum and not guaranteeing a feasible point at the end of the search.

However, if the comparison is only between *AQSD* and *Mod.AQSD*, then with a modified stopping criteria that stops the search in the box if the convergence rate is very slow in *Mod.AQSD* has an advantage and a lower value of **r**. The number of boxes explored by *AQSD* is on average lower than the *Mod.AQSD* due to the modified stopping criteria of the *Mod.AQSD*.

BRSP and *Mod.BRSP* algorithms score the best values of **r** in almost all the algorithms. Separating the penalty values from the objective function and prioritizing the points with lower penalty value over lower objective function value proves to be a better approach to find the box's local minimum.

The average number of boxes checked by the *BRSP* is higher than *Mod. BRSP*. Since the potential of finding a feasible point is higher inside the box due to the strength of the procure box pruning algorithms. Then the boxes checked by *Mod.BRSP* have higher possibility to return a feasible point, allowing the procure search algorithm to rearrange the boxes and branch further.

When the optimization problem is constraints-free, as in *denschnd* and *biggs5*, *BRSP* has lower error value **r** or better timing than *Mod.BRSP*. This can be explained that since there are no restrictions over the variables domains' in the optimization problem, therefore, it is more safe to go outside a box b during the local search. This is only in case the point x_{k+1} outside of b is having a lower penalty value or objective function value as it is shown in the algorithm `isBetterPoint(`x_1,x_2`)` in figure 4.14.

In conclusion, after running the tests and the benchmarks, we can conclude that the box ratio with separated penalty algorithm is better with respect to time taken to find the local minimum than the Armijo rule with quadratic penalty steepest descent algorithm. Moreover, using a random algorithm to try to find the local minimum is a faster technique than calculating the mid-point of the box as it increases the chances of finding a better feasible point.

6

Conclusion and Future work

In this work, we propose to solve optimization problems with a hybrid of continuous constraint satisfaction and local search. We introduced two local search algorithms and integrated them into the continuous domain constraint framework Procure.

In CCSP, the search space is divided into several areas, called boxes. Then a search algorithm is performed on these boxes to find a feasible point inside every box. Two local search algorithms were introduced to override Procure's technique to search inside the box which is simply returning the midpoint of the box and focusing more on the constraint propagations and branching and pruning.

The first algorithm is Armijo rule with quadratic penalty steepest descent (*AQSD*). It uses the steepest descent technique with the famous Armijo rule for selecting the step length and quadratic penalty function in order to integrate the constraints into the objective function. The local search algorithm stops when a counter decreases to zero after a number of iteration with no point x_{k+1} having a better quadratic penalty function value than x_k.

However, from the tests we conducted, we found that the algorithm takes a very long time to converge to a local minimum. Therefore, in the *Mod.AQSD*, we altered to stopping criteria such that the counter will stop decreasing if the difference between x_{k+1} and x_k is significant. This improved the time scored for the search but the improvement is not very significant. Specially when it is compared to a simple random search algorithm inside the box.

The second algorithm we introduced is the Box ratio with Separate Penalty steepest descent (*BRSP*). It also uses the steepest descent technique. For the step length it uses a greedy algorithm that tries the largest step to the edge of the box. The constraints are not integrated in the objective function. However, steepest descent is applied over the

violated constraints at the point x_k. The algorithm does not force any boundaries for x_k to be inside the box b.

A modification is applied to the *BRSP* algorithm. This modification (*Mod.BRSP*) forces the algorithm to select points only from inside the given box b. The algorithm showed its efficiency in the examples discussed in this work. Having the fast times in comparison with the other algorithms.

We tested the algorithms introduced, their modification, a random selection algorithm and Procure's default midpoint algorithm on a set of benchmarks. The results obtained shows that:

- *BRSP* technique for local search inside the box have the fastest convergence rate to a local minimum among the other algorithms.
- *Mod.BRSP* algorithm is generally the fastest local search algorithm in comparison with the other discussed algorithms. This can be explained by the strong box pruning algorithms executed by Procure over the box. Hence, restricting the point to be inside the box interval is more efficient.
- The algorithm with slowest convergence rate is the *AQSD* as it takes a relatively longer time in a single box to converge to a local minimum.
- *BRSP* have a slight advantage over *Mod.BRSP* in unconstrained optimization problems. Since allowing the algorithm to converge to a local minimum outside the box b starting inside b will return a feasible point. This is because there is no restriction on the domain of the optimization problem.

We suggest for future work to improve the *BRSP* algorithm and to apply different alternative local search techniques.

BRSP algorithm can be improved by making the value **r** dynamic, such that its weight varies according to the amount of violated constraints at the selected point x_k. Moreover, we assign the number of $restarts$ to 50. We would suggest to make this number dynamic with respect to the size of the split boxes so it would be proportional to the volume of the box.

We would also suggest for future work to investigate more in applying different local search techniques other than the steepest descent algorithms like Lagrangian methods and compare the results with the ones obtained in this work.

Bibliography

[1] G Arfken. "The method of steepest descents". In: *Mathematical methods for physicists* 3 (1985), pp. 428–436.

[2] L. Armijo et al. "Minimization of functions having Lipschitz continuous first partial derivatives". In: *Pacific Journal of mathematics* 16.1 (1966), pp. 1–3.

[3] F Benhamon, D McAllester, and P Van Hentenryck. "CLP (Intervals) revisited". In: *Rapport technique, Citeseer* (1994), p. 30.

[4] F. Benhamou. "Heterogeneous constraint solving". In: *Algebraic and Logic Programming*. Springer. 1996, pp. 62–76.

[5] P. T. Boggs and J. W. Tolle. "Sequential quadratic programming". In: *Acta numerica* 4 (1995), pp. 1–51.

[6] R. Chelouah and P. Siarry. "Tabu search applied to global optimization". In: *European Journal of Operational Research* 123.2 (2000), pp. 256–270.

[7] A. R. Conn, N. I. Gould, and P. L. Toint. *Trust region methods*. Vol. 1. Siam, 2000.

[8] J. Cruz. "Constraint Reasoning for Differential Models". Pedro Barahona (superv.); PhD thesis. FCT/UNL, 2003.

[9] G. B. Dantzig. "Linear Programming". In: *Operations Research* 50.1 (2002), pp. 42–47.

[10] E. Davis. "Constraint propagation with interval labels". In: *Artificial intelligence* 32.3 (1987), pp. 281–331.

[11] A. Dekkers and E. Aarts. "Global optimization and simulated annealing". In: *Mathematical programming* 50.1-3 (1991), pp. 367–393.

[12] J. E. Dennis Jr and R. B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Vol. 16. Siam, 1996.

[13] L. Di Gaspero, J. Gärtner, N. Musliu, A. Schaerf, W. Schafhauser, and W. Slany. "A hybrid LS-CP solver for the shifts and breaks design problem". In: *Hybrid Metaheuristics*. Springer, 2010, pp. 46–61.

[14] G Di Pillo and L Grippo. "A new augmented Lagrangian function for inequality constraints in nonlinear programming problems". In: *Journal of Optimization Theory and Applications* 36.4 (1982), pp. 495–519.

[15] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A mathematical programming language*. AT&T Bell Laboratories Murray Hill, NJ 07974, 1987.

[16] M. Gendreau and J.-Y. Potvin. *Handbook of metaheuristics*. Vol. 2. Springer, 2010.

[17] P. E. Gill, W. Murray, and M. H. Wright. *Practical optimization*. Academic Press, 1981, pp. I–XVI, 1–401. ISBN: 978-0-12-283952-8.

[18] L. Granvilliers and F. Benhamou. "Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques". In: *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006), pp. 138–156.

[19] E. Hansen and G. W. Walster. *Global optimization using interval analysis: revised and expanded*. Vol. 264. CRC Press, 2003.

[20] P. V. Hentenryck and L. Michel. *Constraint-based local search*. The MIT Press, 2009.

[21] M. R. Hestenes. "Multiplier and gradient methods". In: *Journal of optimization theory and applications* 4.5 (1969), pp. 303–320.

[22] J. Hooker. *Logic-based methods for optimization: combining optimization and constraint satisfaction*. Vol. 2. John Wiley & Sons, 2011.

[23] R. Horst, P. M. Pardalos, and H. E. Romeijn. *Handbook of global optimization*. Vol. 2. Springer, 2002.

[24] E. Hyvönen. "Constraint reasoning based on interval arithmetic: the tolerance propagation approach". In: *Artificial Intelligence* 58.1 (1992), pp. 71–112.

[25] L. V. Kantorovich. "A new method of solving some classes of extremal problems". In: *Doklady Akad Sci USSR*. Vol. 28. 1940, pp. 211–214.

[26] N. Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Combinatorica* 4.4 (1984), pp. 373–396.

[27] R. B. Kearfott. "Interval computations: Introduction, uses, and resources". In: *Euromath Bulletin* 2.1 (1996), pp. 95–112.

[28] O. Lhomme. "Consistency techniques for numeric CSPs". In: *IJCAI*. Vol. 93. Citeseer. 1993, pp. 232–238.

[29] H. R. Lourenço, O. C. Martin, and T. Stutzle. "Iterated local search". In: *arXiv preprint math/0102188* (2001).

[30] A. K. Mackworth. "Consistency in networks of relations". In: *Artificial intelligence* 8.1 (1977), pp. 99–118.

[31] Z. Michalewicz. *Genetic algorithms+ data structures= evolution programs*. springer, 1996.

[32] J. Mockus. "Global Optimization and the Bayesian Approach". In: *Bayesian Approach to Global Optimization*. Springer, 1989, pp. 1–3.

[33] U. Montanari. "Networks of constraints: Fundamental properties and applications to picture processing". In: *Information sciences* 7 (1974), pp. 95–132.

[34] R. E. Moore. *Interval analysis*. Vol. 2. Prentice-Hall Englewood Cliffs, 1966.

[35] J. J. Moré and D. C. Sorensen. "Computing a trust region step". In: *SIAM Journal on Scientific and Statistical Computing* 4.3 (1983), pp. 553–572.

[36] J. J. Moré and D. J. Thuente. "Line search algorithms with guaranteed sufficient decrease". In: *ACM Transactions on Mathematical Software (TOMS)* 20.3 (1994), pp. 286–307.

[37] P. M. Morse and H. Feshbach. "Asymptotic series; method of steepest descent". In: *Methods of Theoretical Physics, Part I* (1953), pp. 434–443.

[38] W. J. Older and A. Vellino. "Constraint Arithmetic on Real Intervals." In: *WCLP*. 1991, pp. 175–195.

[39] M. J. Powell. *" A method for non-linear constraints in minimization problems"*. UKAEA, 1967.

[40] J.-F. Puget and P. Van Hentenryck. "A constraint satisfaction approach to a circuit design problem". In: *Journal of global optimization* 13.1 (1998), pp. 75–93.

[41] D. Sam-Haroud and B. Faltings. "Consistency techniques for continuous constraints". In: *Constraints* 1.1-2 (1996), pp. 85–118.

[42] P. F. for Science and Technology. *Project PROCURE: Probabilistic Constraints for Uncertainty Reasoning in Science and Engineering Applications*. July 2014. URL: http://centria.di.fct.unl.pt/projects/procure/index.html.

[43] P. Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *Principles and Practice of Constraint ProgrammingCP98*. Springer, 1998, pp. 417–431.

[44] G. Sidebottom and W. S. Havens. "Hierarchical arc consistency for disjoint real intervals in constraint logic programming". In: *Computational Intelligence* 8.4 (1992), pp. 601–623.

[45] T. Steihaug. "The conjugate gradient method and trust regions in large scale optimization". In: *SIAM Journal on Numerical Analysis* 20.3 (1983), pp. 626–637.

[46] P. Van Hentenryck, D. McAllester, and D. Kapur. "Solving polynomial systems using a branch and prune approach". In: *SIAM Journal on Numerical Analysis* 34.2 (1997), pp. 797–827.

[47] P. J. Van Laarhoven and E. H. Aarts. *Simulated annealing*. Springer, 1987.

[48] R. J. Vanderbei. *Nonlinear Optimization Models*. July 2014. URL: http://orfe.princeton.edu/~rvdb/ampl/nlmodels.

[49] D. L. Waltz. *Generating semantic description from drawings of scenes with shadows*. 1972.

[50] S. Wright and J Nocedal. *Numerical optimization*. Vol. 2. Springer New York, 1999.

[51] S. J. Wright. "Continuous Optimization (Nonlinear and Linear Programming)". In: *Foundations of Computer-Aided Process Design* (1999).